VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE APLIKACÍ NA GPU V JAZYCE PYTHON

ACCELERATION OF PYTHON APPLICATIONS ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MATEJ TURCEL

VEDOUCÍ PRÁCE
SUPERVISOR

ING. MARTA ČUDOVÁ

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Turcel Matej**
Obor: Informační technologie
Téma: **Akcelerace aplikací na GPU v jazyce Python**
Acceleration of Python Applications on GPU
Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s programovacím jazykem Python a jeho knihovnami numpy, pyCUDA a matplotlib.
2. Seznamte se s paradigmaty paralelního programování a s jazykem CUDA.
3. Navrhněte a implementujte sadu mikrotestů s cílem osvojit si základy jazyka CUDA.
4. Zvolte vhodný problém, který budete paralelizovat, vytvořte jeho paralelní řešení v jazyce Python a ověřte jeho funkčnost na GPU.
5. Ověřte výkonnost navržených řešení pomocí běžných metrik.
6. Zhodnoťte a diskutujte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Čudová Marta, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Konvenčne sa v oblasti high performance computing (HPC) používajú prekladané jazyky, ako napríklad C++. Skriptovacie jazyky ako Python sú však pohodlnejšie a vývoj aplikácií je v nich rýchlejší a jednoduchší. Táto práca porovnáva jazyky C++ a Python z hľadiska možnosti akcelerácie výpočtov na grafickej karte. Jej cieľom je ukázať, že skriptovacie jazyky sú taktiež použiteľné na implementáciu HPC aplikácií a poukázať na ich výhody a nevýhody oproti prekladaným jazykom. Za týmto účelom je implementovaných niekoľko programov. Tie pozostávajú z niekoľkých menších testovacích programov a jedného väčšieho programu, riešiaceho výpočtovo náročný problém. Implementácie týchto programov v jazykoch C++ a Python sú porovnané ako z hľadiska výkonu, tak z hľadiska náročnosti implementácie.

Abstract

Compiled languages, such as C++, are conventionally used in the field of high performance computing (HPC). However, scripting languages like Python are more convenient and application development is quicker and simpler in these languages. This work compares C++ and Python in terms of the possibilities of computation acceleration on graphics card. Its aim is to show that scripting languages are also suitable for the implementation of HPC applications, and point out their advantages and disadvantages compared to compiled languages. To this purpose, a number of programs have been implemented. Several smaller programs for testing purposes and a larger one, implementing a computationally intensive problem. The implementations of these programs in C++ and Python are compared in terms of performance, as well as difficulty of implementation.

Kľúčové slová

vysokovýkonné výpočty, akcelerácia výpočtu, grafická karta, akcelerácia na GPU, skriptovacie jazyky, prekladané jazyky, Python, C++, CUDA, PyCUDA, porovnanie výkonu, veľkosť kódu

Keywords

high performance computing, acceleration of computation, graphics card, GPU acceleration, scripting languages, compiled languages, Python, C++, CUDA, PyCUDA, performance comparison, code size

Citácia

TURCEL, Matej. *Akcelerace aplikací na GPU v jazyce Python.* Brno, 2017. 44 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Čudová Marta, Ing.

Akcelerace aplikací na GPU v jazyce Python

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Marty Čudovej. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……
Matej Turcel
15. mája 2017

Poďakovanie

Rád by som poďakoval vedúcej mojej práce, Ing. Marte Čudovej, za pomoc, pripomienky a námety pri tvorbe práce.

Táto práca bola podporená Ministerstvom školstva, mládeže a telovýchovy Českej republiky z projektu Veľké infraštruktúry pre výskum, experimentálny vývoj a inovácie „IT4Innovations národné superpočítačové centrum – LM2015070“.

Obsah

1 Úvod

Táto práca sa venuje akcelerácii výpočtovo náročných aplikácií na grafickej karte (ďalej len *GPU)*. Hlavným zameraním je porovnanie možností akcelerácie na GPU pre rôzne platformy, konkrétne porovnanie medzi prekladanými a skriptovacími programovacími jazykmi. Konvenčne sa v oblasti vysokovýkonných výpočtov (*HPC – „high performance computing“*) používajú prekladané jazyky. Skriptovacie jazyky sú však užívateľsky pohodlnejšie, vďaka čomu sú v dnešnej dobe veľmi populárne.

Cieľom tejto práce je ukázať, že skriptovacie jazyky sú taktiež použiteľné na implementáciu HPC aplikácií a poukázať na ich výhody a nevýhody oproti prekladaným jazykom. Motiváciou je jednoduchosť skriptovacích jazykov, ako z hľadiska učenia, tak z hľadiska používania. S tým tiež súvisí ich vysoká a stále rastúca popularita a rozšírenosť. V tejto práci je skupina prekladaných jazykov zastúpená jazykmi C/C++. Zo skupiny skriptovacích jazykov bol vybraný jazyk Python, predovšetkým kvôli jeho rozšírenosti a popularite.

Skriptovacie jazyky umožňujú vytvárať aplikácie rýchlo a jednoducho. Rozsah ich zdrojového kódu býva v porovnaní s prekladanými jazykmi výrazne menší. Cenou je však o niečo nižší výkon. To je pri akcelerácii na GPU akceptovateľné, keďže samotný výpočet na GPU je rovnako výkonný, ako pri použití prekladaných jazykov. Je to preto, lebo výpočet na GPU je implementovaný oddelene od obslužného kódu. Práve tento výpočet na GPU väčšinou tvorí podstatnú časť trvania programu. Obslužný kód v jazyku C++ alebo Python má často zanedbateľný vplyv na celkový výkon aplikácie. To platí obzvlášť v prípade, keď je na GPU akcelerovaný veľký, náročný problém.

V jazykoch C++ a Python je v rámci tejto práce implementovaných niekoľko mikrotestov, ktoré slúžia na oboznámenie sa s rozhraním pre programovanie aplikácií na GPU. V jazyku C++ toto rozhranie poskytuje natívna knižnica CUDA, v jazyku Python je poskytované knižnicou PyCUDA. Mikrotesty porovnávajú rôzne parametre pre jednotlivé jazyky, predovšetkým výkon implementácií. Dôležitá je však aj náročnosť implementácie či rozsah zdrojového kódu.

Ďalej je implementovaný komplexnejší problém, ktorý patrí do kategórie výpočtovo náročných problémov vhodných na akceleráciu na GPU. Jedná sa o problém ruksaku, ktorý je riešený pomocou genetického algoritmu. Tento problém bol mimo tejto práce implementovaný v jazyku C++. V rámci tejto práce bol obslužný kód prepísaný do jazyka Python, pričom kód vykonávaný na GPU ostal prevažne zachovaný. Táto práca obsahuje návrh a popis prekladu programu do jazyka Python, ako aj porovnanie implementácií v oboch jazykoch. Sú popísané problémy, ktoré boli riešené počas prepisovania a taktiež ich riešenia. Porovnanie implementácií sa sústredí ako na výkon, tak na náročnosť implementácie a rozsah zdrojového kódu.

2 Úvod do problematiky

V tejto kapitole sú popísané technológie používané pri tvorbe aplikácií využívajúcich grafickú kartu. Je tu stručne popísané použitie superpočítača obsahujúceho mnoho výpočtových uzlov s grafickými kartami. Taktiež sú popísané možnosti akcelerácie výpočtov pomocou GPU v programovacom jazyku Python, ktorého základné črty sú tiež stručne popísané.

Okrem technológie CUDA, ktorá bude popísaná neskôr, je možné výpočty akcelerovať aj inými spôsobmi. Programovací model MPI (*Message Passing Interface*) napríklad slúži na koordináciu výpočtov v distribuovaných systémoch. Tento model nebude bližšie popisovaný. Existuje niekoľko implementácií tohto modelu, napríklad OpenMPI [1]. Ďalšou možnosťou je platforma OpenCL [2], ktorá slúži hlavne na koordináciu distribuovaných a paralelných výpočtov na rôznych zariadeniach (CPU, GPU, FPGA,...). Táto platforma taktiež nebude bližšie popisovaná.

Ďalej je popísaný genetický algoritmus a problém ruksaku. V rámci tejto práce je implementovaný genetický algoritmus, ktorý rieši základný jednodimenzionálny problém ruksaku. Potrebné pojmy sú v tejto kapitole vysvetlené.

2.1 Grafická karta (GPU)

Grafické karty, tiež nazývané *GPU* (ang. *graphics processing unit*), boli pôvodne vyvinuté z potreby akcelerovať výpočty spojené so zobrazovaním predovšetkým 3D objektov a scén, napríklad v počítačových hrách [3]. V dnešnej dobe, vďaka existencii aplikačných rozhraní, napr. CUDA [4], sú však grafické karty použiteľné na rôzne iné účely než hranie hier alebo akceleráciu prehrávania videa. Medzi tieto účely patrí akcelerácia výpočtovo náročných výpočtov, ako napr. Fourierová transformácia alebo rôzne simulácie, napríklad N-body simulácia.

CUDA [4] je platforma umožňujúca použitie grafickej karty z užívateľského programu. Grafická karta musí byť kompatibilná s CUDA architektúrou, teda musí spĺňať určité kritériá. Architektúra CUDA je bližšie popísaná v kapitole 2.1.1. Táto platforma, respektíve architektúra, bola vyvinutá firmou NVIDIA [5], ktorá je tiež výrobcom grafických kariet. Takmer všetky moderné grafické karty firmy NVIDIA tieto kritériá spĺňajú, teda sú s touto architektúrou kompatibilné.

Moderné grafické karty kompatibilné s CUDA architektúrou poskytujú vysoký výkon ťažiaci z paralelizácie výpočtu. Pri vhodnom využití ich možností môžu poskytnúť výpočtový výkon rádovo vyšší než CPU. Takéto grafické karty sú rozšírené v stolných aj prenosných počítačoch a teda sú prístupné veľkému množstvu užívateľov.

Nevýhodou grafických kariet je relatívne nízka rýchlosť prenosu dát medzi GPU a CPU a obecne vysoká cena spustenia programu na GPU. Preto je vhodné na GPU akcelerovať iba tie aplikácie, u ktorých sa to naozaj vyplatí – výpočtovo intenzívne aplikácie. Obvykle je výhodné výpočet akcelerovať na GPU v prípade, keď čas výpočtu je značne dlhší než čas prenosu vstupných a výstupných dát [3]. Je tiež dôležité, aby výpočet samotný nepristupoval k dátam náhodne a jeho tok riadenia bol jednoduchý. Veľké množstvo podmienených vetiev, rozhodovanie počas výpočtu atp. je nežiadúce. Grafické karty totiž nie sú optimalizované na vetvenie toku riadenia, ale na výpočtový výkon, na rozdiel od CPU. CPU naopak obsahuje veľké množstvo rýchlej vyrovnávacej pamäte, prediktor skokov atp. Je teda vhodnejšie prispôsobené pre aplikácie so zložitým tokom riadenia. Jeho výpočtový výkon je však relatívne nízky, aj keď je výpočet optimalizovaný napr. použitím vektorových inštrukcií [3].

Typicky vykonanie výpočtu na GPU pozostáva z niekoľkých krokov [3]:

- alokácia pamäte na GPU,
- prenos vstupných dát do pamäte GPU,
- spustenie výpočtu na GPU,
- prenos výsledkov do operačnej pamäte,
- uvoľnenie alokovaných zdrojov.

2.1.1 Architektúra CUDA

V tejto kapitole je popísaná CUDA architektúra z hľadiska hardvérového a softvérového modelu.

Hardvér

CUDA-kompatibilná grafická karta pozostáva z hlavnej pamäte a niekoľkých výpočtových jednotiek zvaných *Streaming Multiprocessor* (SM) [3]. Obrázok 2.1 znázorňuje architektúru CUDA z hľadiska štruktúry hardvéru.

Hlavná pamäť je rozdelená na globálnu pamäť (všeobecné použitie), pamäť konštánt (iba na čítanie), pamäť textúr (optimalizovaná pre 2D prístup) a lokálnu pamäť vláken [3].

Multiprocesor pozostáva z tzv. zdieľanej pamäte, registrov a niekoľkých CUDA jadier, tiež zvaných *Streaming Processor* (SP) [3]. Kód vykonávaný na jednom SP má prístup k registrom tohto SP, zdieľanej pamäti príslušného SM a ku globálnej, konštantnej a textúrovej pamäti. Prístup ku registrom a k zdieľanej pamäti je spravidla rýchlejší než prístup ku hlavnej pamäti. Dáta presahujúce kapacitu registrov je možné uložiť v tzv. lokálnej pamäti. Tento názov je mierne zavádzajúci, keďže v skutočnosti sa tieto dáta nachádzajú v hlavnej pamäti. Pojem „lokálna pamäť“ označuje časť hlavnej pamäte, ktorá je vyhradená pre jedno konkrétne vlákno. Je teda pomalšia než registre alebo zdieľaná pamäť.

Softvér

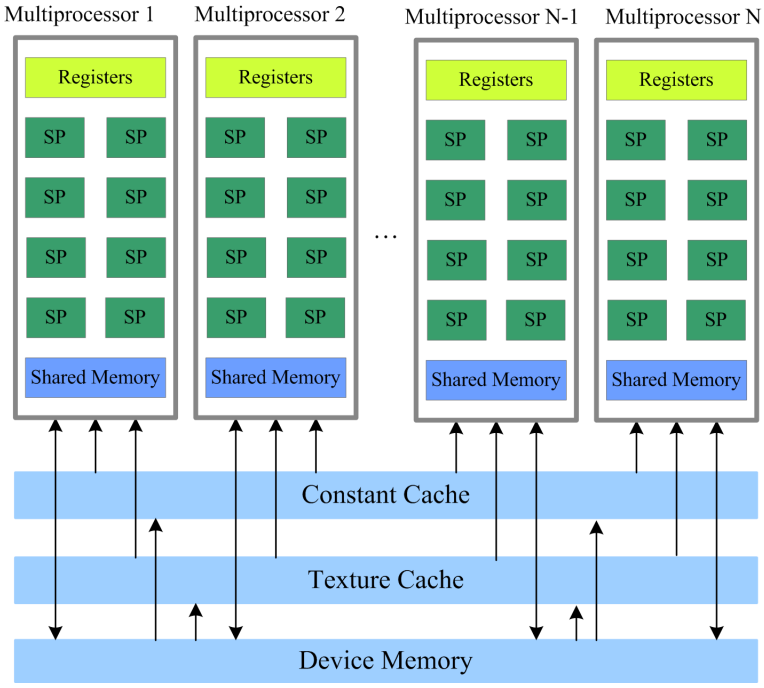
Kód spúšťaný na grafickej karte sa nazýva kernel [3]. Kernel sa vykonáva paralelne vo viacerých vláknach. Jedno vlákno je vykonávané na jednom SP. Vlákna sú združené do menších skupín, tzv. *warpov.* Warp je základnou jednotkou vykonania programu na GPU. Warpy sú združené do väčších skupín, tzv. blokov. Blok je základnou jednotkou alokácie.

Warp typicky pozostáva z 32 vláken [3]. Jeho veľkosť je pevne daná použitým hardvérom. Užívateľský program nemá o konkrétnom warpe informácie (napr. identifikačné číslo, ako je tomu u blokov). K dispozícii je iba veľkosť warpu, ktorá je pre všetky warpy rovnaká. Vlákna v jednom warpe sa vykonávajú paralelne. Warp je vzhľadom k ostatným warpom vykonaný kvázi-paralelne, teda z pohľadu užívateľa sú všetky vlákna vykonané paralelne, ale v skutočnosti tomu tak nemusí byť. Aplikačné rozhranie CUDA poskytuje možnosti synchronizácie vláken v rámci jedného bloku. Naprieč blokmi nie je možné vlákna spoľahlivo synchronizovať. Je to dané spôsobom, akým sú bloky vykonané v prípade, že nemôžu byť všetky vykonané súčasne – najprv je vykonaný jeden blok, až potom je spustený druhý.

Blok je trojdimenzionálna štruktúra združujúca vlákna. Užívateľ určuje jeho rozmery pri spustení kernelu. Jeho maximálne rozmery sú hardvérovo obmedzené, navyše môžu byť tiež obmedzené množstvom dostupnej pamäte. Vlákna v jednom bloku sú vykonané na jednom SM. Majú teda k dispozícii zdieľanú pamäť a je možné ich v rámci toho-ktorého bloku navzájom synchronizovať. Bloky sú združené to tzv. *gridu*.

Grid je trojdimenzionálna štruktúra združujúca bloky. Podobne ako pri bloku, užívateľ určuje jeho rozmery pri spustení kernelu a maximálne rozmery sú obmedzené hardvérom. Všetky bloky

v rámci gridu majú rovnaké rozmery. Jeden kernel obsahuje jeden grid, a teda grid obsahuje všetky bloky kernelu. Bloky v rámci gridu môžu ale nemusia byť vykonané paralelne. Ak počet blokov presahuje počet SM, niektoré bloky sú vykonané sériovo vzhľadom k iným blokom. Gridy, respektíve kernely, sú vykonané sériovo v rámci jedného *streamu* [3]. Stream je sekvencia operácií, ktoré sa na GPU vykonávajú v takom poradí, v akom sú zapísané v riadiacom kóde. Na jednej grafickej karte sa môže súčasne vykonávať niekoľko kernelov v niekoľkých streamoch. V rámci jedného streamu ale beží v jednom momente iba jeden kernel.



Obrázok 2.1: Architektúra CUDA – štruktúra GPU. Obrázok znázorňuje hardvérový model grafických kariet CUDA: multiprocesory, ich pamäť, SP a registre a prístup multiprocesorov ku globálnej pamäti. [6]

2.2 Superpočítač

Superpočítač pozostáva z mnohých navzájom prepojených počítačov, nazývaných výpočtové uzly [7]. Výpočtový uzol obvykle obsahuje procesor, fyzickú pamäť, prípadne ďalšie zariadenia, napríklad grafickú kartu. Superpočítač môže pozostávať z niekoľkých druhov výpočtových uzlov, ktoré sa navzájom líšia svojím vybavením. Obvykle je ale žiaduca homogénnosť, teda veľké množstvo rovnakých uzlov. Na prepojenie uzlov slúži špecializovaná vysokorýchlostná sieť [7].

Pre účely testovania bol v tejto práci použitý superpočítač Anselm nachádzajúci sa vo výpočtovom centre IT4I v Ostrave [8]. Superpočítač Anselm obsahuje 209 výpočtových uzlov, z toho 23 je vybavených grafickými kartami. Každý z týchto 23 uzlov má 96 GB pamäte a jednu grafickú kartu NVIDIA Kepler K20 [9] (5 GB pamäti, špičkový výkon 3,52 TFLOPS v jednoduchej presnosti. Definícia TFLOPS – viď kapitola 2.5).

Na počítači Anselm sa typicky pracuje pomocou SSH prístupu [10]. Počítač používa systém modulov, ktoré obsahujú jednotlivé programy alebo skupiny programov a môžu tiež poskytovať rôzne

verzie jednotlivých programov. Pred prácou s aplikáciami je potrebné načítať moduly obsahujúce požadované aplikácie. Počítač Anselm využíva na vykonávanie jednotlivých výpočtových úloh tzv. *Portable Batch System* (*PBS*). Tento systém zabezpečuje plánovanie úloh a umožňuje ich monitorovanie a správu. Úlohy sú vkladané do front. Pri vytvorení úlohy sa okrem fronty, do ktorej bude úloha zaradená, špecifikujú tiež požiadavky úlohy, ako napr. počet procesorov, maximálna doba vykonávania atď. Fronty môžu mať rôzne priority a môže v nich byť dostupné rôzne vybavenie superpočítača. Napríklad pre úlohy využívajúce GPU existuje na počítači Anselm samostatná fronta „qnvidia“, tzn. GPU sú prístupné iba úlohám vykonávaným v tejto fronte. Úloha je po zaradení do fronty vykonaná asynchrónne, teda prihlásený užívateľ po jej zaradení do fronty a počas jej vykonávania ďalej pracuje s SSH terminálom nezávisle na úlohe. Je možné taktiež „interaktívne prihlásenie“ do určitej fronty. Pri ňom sa špecifikujú parametre tak, ako pri bežnom zaradení do fronty, ale po zaradení do fronty užívateľ počká vo fronte a potom je úloha vykonaná priamo v užívateľovom termináli, tzn. blokujúce vykonanie úlohy. Daná úloha môže byť napríklad program typu „shell“, po spustení ktorého sú užívateľovi dostupné rovnaké prostriedky, ako úlohám vykonávaným v tej-ktorej fronte.

Pre účely testovania bol použitý tiež školský server SC-GPU1[1]. Tento server má nasledujúce parametre:

- 2x CPU Intel® Xeon® CPU E5-2620 v3 – max. 2,4 GHz, 6 jadier, Hyper-threading,
- 64 GB RAM,
- 4x GPU NVIDIA GeForce GTX 1080 [11] – 8GB pamäti, špičkový výkon 8,2 TFLOPS v jednoduchej presnosti.

2.3 Jazyk Python

Konvenčne sa v oblasti HPC používajú jazyky blízke hardvéru, tj. prekladané nízkoúrovňové jazyky – predovšetkým jazyk C, prípadne C++. Tieto jazyky umožňujú detailnejšie ladenie programov, predovšetkým pre výkonové optimalizácie. Nevýhodou však je, že musia byť na každom cieľovom systéme pred použitím preložené a nie sú dostatočne expresívne. Často nútia užívateľa venovať sa detailom, ktoré sú mnohokrát nepodstatné alebo môžu byť automaticky odvodené, na úkor dôležitejších problémov. Ich syntax neumožňuje stručne vyjadriť mnohé často používané konštrukcie. Napríklad iterácia cez prvky poľa v jazyku C je oproti väčšine skriptovacích jazykov syntakticky omnoho rozsiahlejšia, pričom výsledný efekt je rovnaký. To vedie jednak k zvýšenému úsiliu pri tvorbe programov, a taktiež k zníženej čitateľnosti zdrojového kódu. Veľká časť kódu sa venuje úkonom, ktoré by mohli byť vykonávané automaticky. Užívateľ by potom mohol venovať viac úsilia jadru problému, ktorý rieši. To sa týka okrem pokročilých konštrukcií jazyka tiež správy zdrojov, ako je napr. alokácia pamäti, prenos dát medzi CPU a GPU, atď.

Python [12] je dynamický interpretovaný programovací jazyk. Vyznačuje sa čistou a expresívnou syntaxou, vďaka čomu je jednoduché sa ho naučiť aj ho používať. Keďže je interpretovaný, je možné v ňom tvoriť platformovo nezávislé programy. Tie fungujú na rôznych operačných systémoch bez nutnosti prekladu alebo konfigurácie. Ďalšou výhodou je veľké množstvo štandardných aj externých knižníc a rozšírení, ktoré ďalej uľahčujú tvorbu programov v tomto jazyku. Vďaka spomenutým vlastnostiam sa jazyk Python stal veľmi populárnym a rozšíreným, čo je jeho ďalšou výhodou. Oproti jazykom C a C++ je jednoduchšie sa ho naučiť aj ho používať a programy v ňom vytvorené sú jednoduchšie na nasadenie a použitie. Vďaka spomínanej expresívnosti je zdrojový text jazyku Python prehľadnejší a ľahšie sa v ňom orientuje. Nevýhodou je nižšia výkonnosť v porovnaní s prekladanými jazykmi, keďže Python je jazyk interpretovaný. Okrem toho

1 Adresa servera SC-GPU1: `sc-gpu1.fit.vutbr.cz`

je ďalšou hlavnou nevýhodou chýbajúca typová kontrola. To ho činí nevhodným na tvorbu rozsiahlych aplikácií a vedie k horšej udržiavateľnosti kódu.

2.4 Knižnice pre jazyk Python

Pre jazyk Python existuje niekoľko možností použitia GPU na akceleráciu aplikácií. Niektoré možnosti sú popísané v tejto kapitole. Podrobnejšie je popísaná knižnica PyCUDA [13], ostatné knižnice ako napríklad Numba [14] sú popísané iba zbežne.

2.4.1 Knižnica PyCUDA

V jazykoch C/C++ sa pre prácu s grafickou kartou používa knižnica CUDA. Tá poskytuje základné prostriedky pre prácu s GPU, ako je alokácia pamäte na GPU, prenos dát medzi pamäťou CPU a GPU, spúšťanie kernelov, atď.

Pre jazyk Python existuje knižnica PyCUDA [13], ktorá poskytuje rovnakú základnú funkcionalitu ako knižnica CUDA v jazykoch C/C++. Okrem toho poskytuje tiež dodatočnú funkcionalitu uľahčujúcu prácu s GPU. Napríklad trieda `GpuArray` poskytuje rovnaké rozhranie ako trieda `ndarray` knižnice numpy [15], ale dáta sú uložené v pamäti GPU a operácie nad inštanciou tejto triedy sú vykonávané na GPU. Výhodou je okrem zjednodušenia alokácie pamäte a prenosu dát tiež zjednodušenie niektorých základných operácií, napr. sčítanie a násobenie vektorov. Nevýhodou je nižší stupeň kontroly nad automaticky vykonávanými operáciami. Zo zdrojového kódu potom nemusí byť jasné, kedy a ako sa jednotlivé operácie vykonávajú.

Pri použití tejto knižnice musí byť kernel implementovaný v jazyku CUDA C++, ktorý je nadmnožinou C++. Z programu jazyku Python je potom kernel preložený a spustený. Výhodou použitia jazyka Python v kombinácii s knižnicou PyCUDA je predovšetkým zjednodušenie režijných operácií ako je alokácia pamäte a prenos dát. V prípade potreby je možné tiež detailnejšie ladenie a nastavenie parametrov, napríklad spôsob alokácie pamäte, konfigurácia pamäte konštánt a textúr, atď.

2.4.2 Ďalšie možnosti GPU akcelerácie pre jazyk Python

Knižnica PyCUDA predstavuje zjednodušenie oproti natívnej knižnici CUDA. Stále je však relatívne nízkoúrovňová a akcelerácia aplikácií pomocou tejto knižnice je relatívne zložitá. Okrem tejto knižnice existujú tiež iné riešenia, ktoré akceleráciu aplikácií na GPU ešte viac zjednodušujú.

Knižnica Numba [14] napríklad poskytuje možnosť automatickej akcelerácie funkcií v jazyku Python pomocou dekorátorov funkcií. V dekorovaných funkciách sú detekované operácie s objektmi triedy `ndarray` knižnice numpy. Tieto operácie sú v prípadoch, kedy je to vhodné, automaticky vykonané na GPU. Nevýhodou je nedostatok kontroly nad vykonávanými operáciami – operácie nie je možné sledovať ani ovplyvniť. Iné knižnice než PyCUDA neposkytujú dostatok kontroly a ani základné potrebné operácie, napríklad spoľahlivé meranie času pomocou udalostí (tzv. *CUDA events* [16]). Sú určené primárne na použitie už implementovaného, respektíve automaticky vytvoreného akcelerovaného kódu.

2.5 Metriky použité v testoch

Výpočtový výkon je meraný v jednotkách *FLOPS* (ang. *floating-point operations per second*) – počet operácií v plávajúcej desatinnej čiarke za sekundu. Bežne sa používajú násobné jednotky, napr. megaflops (MFLOPS) – milión FLOPS, atď.

Rýchlosť prenosu dát, tzv. priepustnosť, je meraná v jednotkách B/s – počet bajtov prenesených za sekundu, respektíve v násobných jednotkách.

Dôležitá je tiež náročnosť implementácie. Predpokladá sa, že za cenu mierneho zníženia výkonu sa pri použití jazyka Python výrazne zjednoduší implementácia oproti jazykom C a C++. Toto zjednodušenie implementácie je hlavnou motiváciou pre použitie skriptovacích jazykov.

2.6 Genetický algoritmus

Genetické algoritmy sú heuristické postupy určené na hľadanie riešenia zložitých problémov [17]. Medzi takéto problémy patria napríklad problémy, pre ktoré neexistuje exaktné riešenie, alebo problémy, ktorých exaktné riešenie je príliš náročné, napríklad NP-úplné problémy.

Inšpiráciou pre genetické algoritmy sú prírodné evolučné procesy. Genetický algoritmus udržiava niekoľko možných riešení problému, z ktorých pomocou genetických operácií vyberá tie najvhodnejšie. Keďže sa jedná o heuristický postup, výsledné riešenie nemusí byť optimálne [17].

2.6.1 Prvky genetického algoritmu

Každé možné riešenie problému musí byť určitým spôsobom reprezentované ako jedinec [17]. Reprezentácia určitého jedinca sa nazýva chromozóm a môže ju tvoriť napríklad binárny reťazec. Jednotlivé bity tohto reťazca sa nazývajú gény. Hodnoty, ktorých môžu gény nadobúdať, sa nazývajú alely. V prípade binárneho reťazca sú tieto hodnoty 0 a 1.

Pojem populácia označuje množinu všetkých jedincov existujúcich v rámci jednej inštancie genetického algoritmu. V procese genetického algoritmu sa populácia modifikuje pomocou genetických operácií. Inštancia populácie medzi dvoma modifikáciami sa nazýva generácia.

Kvalita jedincov je určená ohodnocovacou funkciou (tzv. *fitness* funkcia). Vstupom tejto funkcie je jedinec. Jej výstupom je číselná hodnota určujúca vhodnosť jedinca, respektíve vhodnosť riešenia, ktoré tento jedinec reprezentuje. Čím vyššia je hodnota ohodnocovacej funkcie, tým je riešenie bližšie optimálnemu riešeniu.

2.6.2 Princíp fungovania

Genetický algoritmus vychádza z počiatočnej množiny jedincov (počiatočnej populácie). Tá je väčšinou vygenerovaná náhodne. V priebehu genetického algoritmu sa počet jedincov v populácii nemení.

Nad touto populáciou sú niekoľko krát vykonávané operácie [17]:

- Evaluácia – ohodnotenie jedincov pomocou ohodnocovacej funkcie. Pre každého jedinca je pomocou ohodnocovacej funkcie vypočítaná jeho kvalita, ktorá korešponduje s kvalitou riešenia reprezentovaného týmto jedincom.
- Selekcia – výber najschopnejších jedincov. Tento výber môže byť čiastočne náhodný a je pri ňom použitá ohodnocovacia funkcia.
- Kríženie – časti chromozómov vybraných jedincov sú zamenené medzi jedincami. Vybraní jedinci použití pri krížení sa nazývajú rodičia. Výsledný jedinec, ktorý vznikol krížením rodičovských chromozómov, sa nazýva potomok.
- Mutácia – časť chromozómu jedinca je náhodne pozmenená.
- Reprodukcia – ponechanie jedinca bez zmien.

Táto sekvencia operácií tvorí jeden evolučný cyklus. Evolučný cyklus sa v rámci evolúcie vykonáva opakovane, dokým nie je dosiahnutá ukončovacia podmienka. Tou môže byť napríklad dosiahnutie požadovanej kvality riešenia, alebo dosiahnute maximálneho počtu generácií.

V priebehu genetického algoritmu sa kvalita populácie postupne zvyšuje pomocou výberu najvhodnejších jedincov a genetických operácií nad nimi. Po ukončení evolúcie je z populácie vybraný najschopnejší jedinec, reprezentujúci najvhodnejšie dosiahnuté riešenie. Toto riešenie môže ale nemusí byť optimálne.

2.7 Problém ruksaku

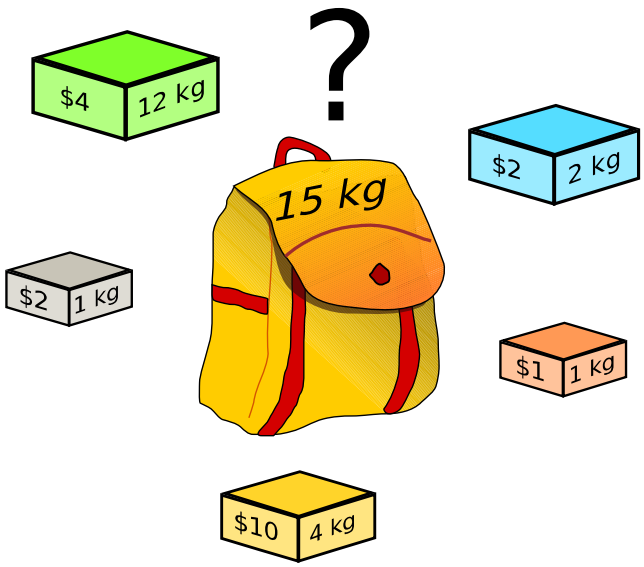
Problém ruksaku (ang. *Knapsack problem* [18]) je NP-úplný [19] problém kombinatorickej optimalizácie. Cieľom tohto problému je maximalizovať zisk vhodným využitím obmedzených zdrojov. V najjednoduchšej variante je toto obmedzenie zdrojov dané iba jedným faktorom, napríklad nosnosťou ruksaku [18].

2.7.1 Neformálny popis

U tohto problému ruksak predstavuje zdroj, ktorého obmedzením je jeho nosnosť. Zisk predstavujú predmety, ktoré je potrebné vložiť do ruksaku. Každý predmet má danú cenu a hmotnosť. Snažíme sa do ruksaku vložiť predmety tak, aby ich celková cena bola čo najvyššia a zároveň ich hmotnosť nepresiahla nosnosť ruksaku.

Základná varianta umožňuje vybrať predmet iba raz. Táto varianta sa nazýva binárny problém ruksaku, alebo tiež 0-1 problém. Ďalšie varianty umožňujú napríklad zvoliť každý predmet ľubovoľný počet krát, alebo je počet výskytov jednotlivých predmetov zhora ohraničený.

U základného jednodimenzionálneho problému ruksaku majú predmety danú iba hmotnosť a cenu. Ich objem nie je braný v úvahu. Taktiež nie je braný v úvahu objem ruksaku, teda v rámci riešenia tohto problému je objem ruksaku neobmedzený. Prvky tohto problému sú znázornené na obrázku 2.4.



Obrázok 2.4: Problém ruksaku. Predmety predstavujú zisk, ktorý je potrebné maximalizovať využitím obmedzeného zdroja. Ten predstavuje ruksak, ktorý má obmedzenú nosnosť. [20]

2.7.2 Formálny popis

Daná je *n*-tica kladných cien predmetov $\langle p_1, p_2, ..., p_n \rangle$, *n*-tica kladných hmotností predmetov $\langle w_1, w_2, ..., w_n \rangle$ a maximálna nosnosť ruksaku $W > 0$.

Cieľom je nájsť podmnožinu predmetov $T \subseteq \{1,2,...,n\}$, ktorá maximalizuje $\sum_{i \in T} p_i$ pri dodržaní podmienky $\sum_{i \in T} w_i \leq W$.

2.7.3 Praktické použitie

Tento problém sa často vyskytuje v praxi a preto majú algoritmy s ním súvisiace široké uplatnenie. Je to tiež dôvodom, prečo existuje mnoho rôznych špecializovaných či generalizovaných variánt tohto problému. Medzi tieto varianty patrí napríklad varianta s viacerými obmedzeniami alebo s niekoľkými ruksakmi [18].

Riešenie tohto problému je aplikovateľné v takých situáciach, kde je potrebné efektívne využiť obmedzené zdroje. Samotný neformálny popis problému, ako aj jeho názov, popisuje jedno takéto uplatnenie. Napríklad v prípade živelnej pohromy musíme opustiť obydlie a môžeme si so sebou vziať iba najpotrebnejšie veci. Tie chceme zabaliť do ruksaku. Vecí ale máme viac, než ruksak unesie a preto z nich musíme vybrať tie najcennejšie.

Klasický problém ruksaku má využitie v mnohých oblastiach, obzvlášť pri procese rozhodovania. Napríklad pri spracovaní surových materiálov je potrebné z kusu materiálu vyrezať čo najviac kusov výrobkov s čo najmenším množstvom odpadu. Nevyužitý materiál tvoriaci medzeru medzi dvoma výrezmi korešponduje s nevyužitou kapacitou ruksaku [18]. Tento problém je prevediteľný na problém rezania materiálu [21], ktorý presnejšie vystihuje jeho podstatu.

Ďalším podstatným odvetvím, v ktorom sa tento problém vyskytuje, je odvetvie financií. Problém ruksaku nastáva pri výbere investícií a portfólií [18]. Investor má obmedzený rozpočet a má k dispozícii niekoľko možných investícií. Investícia má náklady a predpokladaný zisk. Investor chce maximalizovať zisk, pričom celkové náklady nesmú prekročiť jeho rozpočet. To korešponduje s problémom ruksaku, kde predmety sú ekvivalentom investícií a nosnosť ruksaku je ekvivalentom rozpočtu. U reálneho rozhodovania takéhoto druhu je ale potrebné zvážiť tiež risk, čím sa problém komplikuje. V princípe ale tento problém súvisí s alokáciou zdrojov, rovnako ako problém ruksaku.

Problém ruksaku bol tiež využívaný v niektorých kryptosystémoch [18], ktorých bezpečnosť spočívala v náročnosti riešenia tohto problému. Tieto systémy sú však prelomiteľné, takže v dnešnej dobe nie sú veľmi používané.

2.7.4 Možnosti implementácie

Existuje mnoho exaktných metód riešenia problému ruksaku. Jednou z týchto metód je riešenie pomocou dynamického programovania [18]. Ďalším, o niečo efektívnejším prístupom je tzv. „branch and bound“ [22]. Existujú tiež hybridné algoritmy, ktoré kombinujú tieto prístupy.

Vzhľadom na to, že tento problém je NP-úplný, žiadna z exaktných metód nie je dostatočne efektívna. Neexistuje exaktné riešenie s polynomickou časovou náročnosťou. To robí tento problém vhodným na použitie rôznych približných metód. Riešenia získané týmito metódami nemusia byť optimálne, ale v praxi je málokedy potrebné nájsť optimálne riešenie. Väčšinou postačuje riešenie určitej dostatočnej kvality, i keď nie je optimálne. Nájsť takéto riešenia je spravidla jednoduchšie, než nájsť optimálne riešenie.

Jednou z približných metód riešenia tohto problému je polynomiálna aproximačná schéma [18]. Tá zaručuje riešenie, ktorého kvalita je s určitou odchýlkou rovná kvalite optimálneho riešenia. Jej časová a priestorová náročnosť môže byť ale stále príliš vysoká. Taktiež implementácia môže byť relatívne zložitá.

Ďalšou možnosťou je použitie genetického algoritmu. Na aplikáciu takéhoto algoritmu je potrebné vhodným spôsobom reprezentovať riešenie problému ako chromozóm. V binárnom probléme ruksaku môže byť predmet vybraný maximálne raz, teda na reprezentáciu vybraných predmetov postačí binárny reťazec. Jednotka reprezentuje že predmet vybraný je, nula znamená že vybraný nie je. Je tiež potrebná ohodnocovacia funkcia. Tou je celková cena predmetov v ruksaku, teda súčet cien všetkých predmetov, ktoré sú vybrané. Implementácia pomocou genetického algoritmu je teda celkom priamočiara.

3 Návrh a implementácia mikrotestov

Za účelom oboznámenia sa s výkonovými charakteristikami a porovnania výkonu aplikácií vytvorených v jazykoch CUDA C++ a Python bolo vytvorených niekoľko jednoduchých testov:

- prenos dát medzi pamäťou CPU a GPU (kapitola 3.1),
- operácia SAXPY (kapitola 3.2),
- násobenie matíc (kapitola 3.3).

Testy sledujú rozdiely v implementácii GPU akcelerácie v jazykoch C/C++ a Python. Hlavným zameraním je výkon aplikácie, pričom sa okrem výpočtového výkonu, resp. trvania samotného výpočtu, sleduje tiež rýchlosť prenosu dát medzi CPU a GPU.

Všetky vykonané testy merajú rýchlosť prenosu alebo výpočtu, preto je potrebné presné a spoľahlivé meranie času. V testoch bol čas meraný použitím CUDA events [16], ktoré zabezpečujú zaznamenanie počiatočného a konečného času v správnych momentoch. Na meranie času pri použití GPU nie je vhodné použiť čas procesoru preto, lebo vykonávanie úloh na GPU je asynchrónne vzhľadom k CPU. Mohol by tak nastať prípad, že po zaznamenaní času by došlo k prepnutiu úlohy ešte pred spustením operácie na GPU. Použitie časovača GPU pomocou CUDA events je spoľahlivejšie a presnejšie.

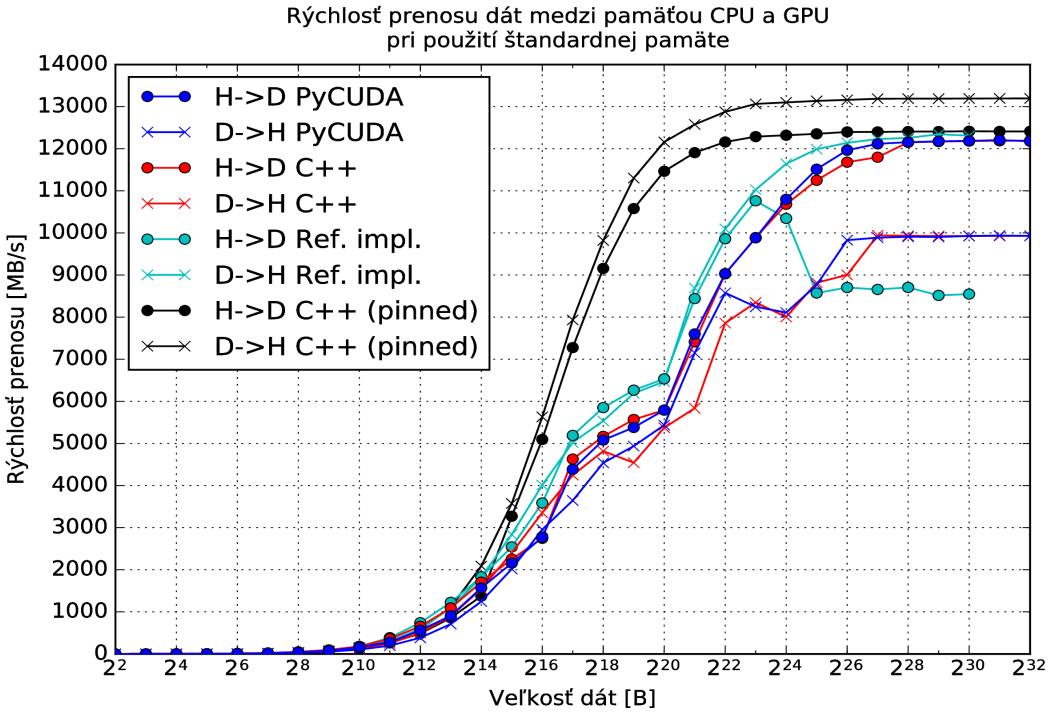
Testy boli vykonané na školskom výpočtovom serveri (popis viď kapitola 2.2). Pri testovaní bola použitá iba jedna grafická karta pripojená na PCI zbernicu procesoru, na ktorom bola úloha spustená.

3.1 Prenos dát medzi pamäťou CPU a GPU

Tento test je zameraný na meranie rýchlosti prenosu dát z operačnej pamäte do pamäte grafickej karty a naopak. Knižnica CUDA podporuje alokáciu tzv. „pinned“ pamäte, tiež nazývanej „page-locked“ pamäť. Pri použití tejto pamäte sú stránky pamäte „zamknuté“, v dôsledku čoho nemôžu byť odložené do „swap“ priestoru na disku. To má za následok zrýchlenie prenosu medzi pamäťou CPU a GPU.

V tomto teste sú porovnané rýchlosti prenosu v závislosti na veľkosti prenášaných dát v jazykoch C++ a Python. Pri každom jazyku sú použité dva spôsoby alokácie („pinned“, „non-pinned“) a pre každý spôsob je meraná rýchlosť v dvoch smeroch (z CPU na GPU a naopak).

Grafy 3.1 a 3.2 ukazujú, že pri použití „pinned“ pamäte sú rýchlosti výrazne vyššie (až o 50 %) a stabilnejšie, ako pri použití štandardnej pamäte. Najväčšie rozdiely sú v rozmedzí veľkosti dát cca. od 64 KB do 16 MB. Pri veľkostiach nad 16 MB sa pri použití štandardnej pamäte rýchlosti prenosu z CPU na GPU blížia rýchlostiam „pinned“ pamäte. Rýchlosti prenosu opačným smerom, tj. z GPU na CPU, sú výrazne pomalšie. Pri tomto prenose je „pinned“ pamäť rýchlejšia o cca. 30 %. Nižšie rýchlosti sú zrejme spôsobené výpadkom zapisovaných stránok z vyrovnávacej pamäte.



Graf 3.1: Porovnanie rýchlostí prenosu z CPU na GPU (*H->D*) a z GPU na CPU (*D->H*) v jazykoch Python (knižnica PyCUDA) a C++ (knižnica CUDA) s využitím štandardnej pamäte, vzhľadom k veľkosti prenášaných dát.

Rýchlosti prenosu použitím natívnej knižnice CUDA v jazyku C++ a knižnice PyCUDA v jazyku Python sú porovnateľné. Pri použití štandardnej pamäte je rýchlosť verzie Python priemerne o 0,02 % vyššia než u verzie C++. Tieto mierne vyššie rýchlosti sú zrejme spôsobené celkovou nestabilitou rýchlosti prenosu pri použití štandardnej pamäte. Pri použití „pinned“ pamäte je verzia Python priemerne o 2,38 % pomalšia. Tento rozdiel je zrejme spôsobený režijnými nákladmi.

Použitie „pinned“ pamäte má najväčší vplyv na rýchlosť prenosu z GPU na CPU. Pri použití štandardnej pamäte je pri veľkostiach nad 4 MB prenos z GPU na CPU výrazne pomalejší než prenos opačným smerom. Pri použití „pinned“ pamäte je však stabilne rýchlejší približne o 8-9 %.

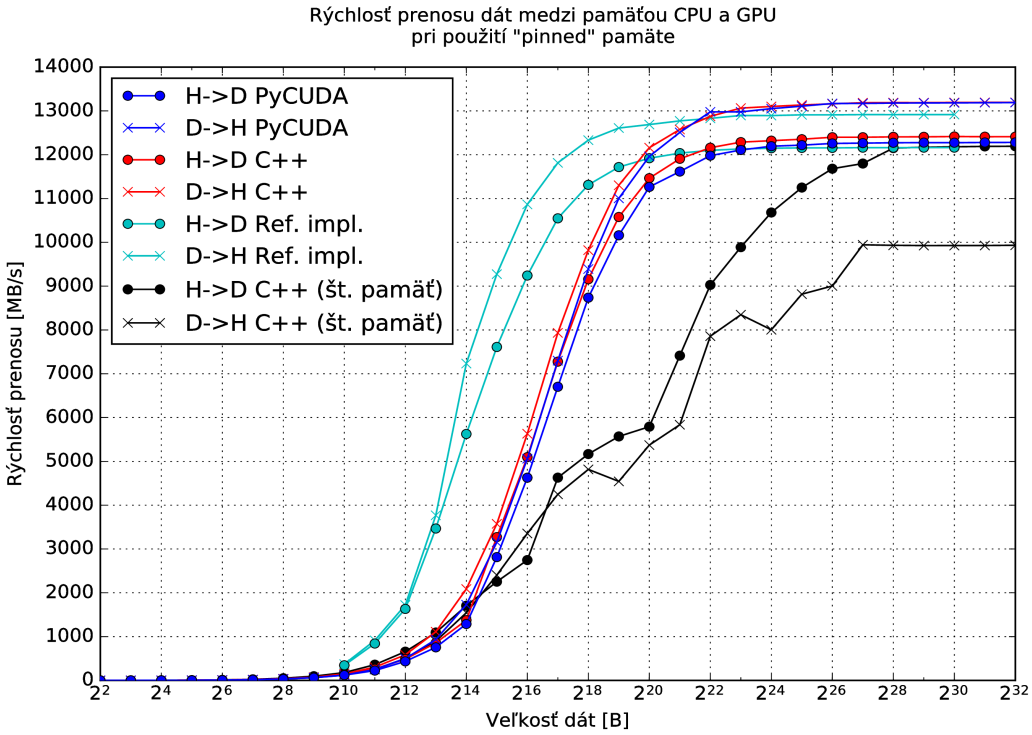
Z výsledkov tohto testu možno usúdiť, že pri prenose dát medzi CPU a GPU je výhodné použiť „pinned“ pamäť, pretože jej použitie zvyšuje priemernú rýchlosť aj stabilitu rýchlosti prenosu.

Namerané rýchlosti približne korešpondujú s rýchlosťami dosahovanými referenčným programom. Ako referenčný program bol použitý program `bandwidthTest` z kolekcie ukážkových programov „CUDA Samples“ [23] implementovaný v jazyku C++.

Priemerná rýchlosť prenosu dát o veľkosti 1 KB až 1 GB:

- s použitím „pinned“ pamäte:
  - referenčná implementácia: 9716,9 MB/s,
  - jazyk C++: 8903,5 MB/s (relatívna odchýlka -8,4 %),
  - jazyk Python: 8726,9 MB/s (relatívna odchýlka -10,2 %);
- s použitím štandardnej pamäte:
  - referenčná implementácia: 6528,4 MB/s,
  - jazyk C++: 6484,6 MB/s (relatívna odchýlka -0,67 %),
  - jazyk Python 6504,2 MB/s (relatívna odchýlka -0,37 %).

Veľké odchýlky pri použití „pinned“ pamäte sú spôsobené tým, že v referenčnej implementácii je použitý odlišný spôsob kopírovania dát než v testovacích programoch. V testovacích programoch je použitá funkcia `cudaMemcpy`, kým v referenčnej implementácii je použitá funkcia `cudaMemcpyAsync`. Rozdiely môže spôsobovať tiež „zahrievacie“ kopírovanie, ktoré je v referenčnej implementácii vykonané pred zahájením merania.



Graf 3.2: Porovnanie rýchlostí prenosu z CPU na GPU (*H->D*) a z GPU na CPU (*D->H*) v jazykoch Python (knižnica PyCUDA) a C++ (knižnica CUDA) s využitím „pinned“ pamäte, vzhľadom k veľkosti prenášaných dát.

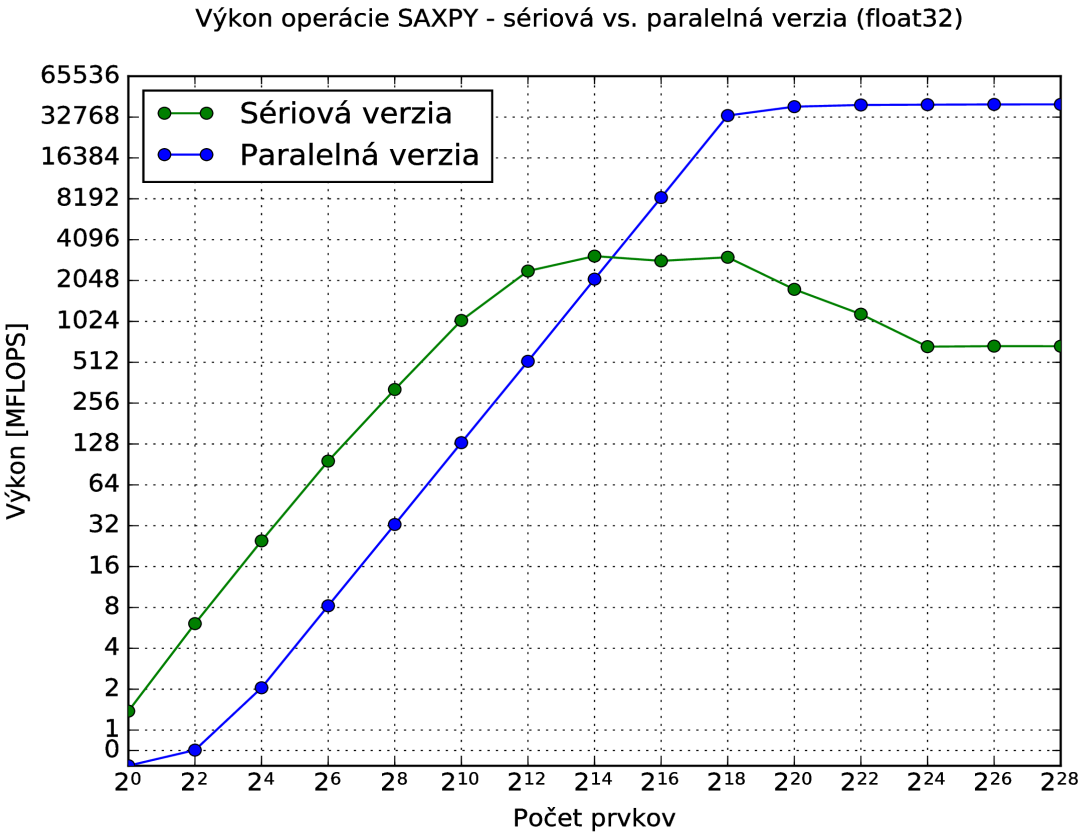
3.2 Operácia SAXPY

Skratka SAXPY (ang. *Single-precision A * X Plus Y*) označuje operáciu *a*X+Y*, kde *a* je skalár a *X*, *Y* sú vektory rovnakej veľkosti. Číselné hodnoty sú uložené ako čísla s plávajúcou desatinnou čiarkou v jednoduchej presnosti. Asymptotická časová náročnosť je *O(n)*, kde *n* je počet prvkov vstupného vektoru. Tento test bol implementovaný iba v jazyku Python. Jedným cieľom tohto testu je porovnanie výkonu sériovej a paralelnej verzie. Druhým je porovnanie času potrebného na prenos dát a na samotný výpočet.

Graf 3.3 znázorňuje výkon sériovej a paralelnej verzie vzhľadom k počtu prvkov vstupných vektorov. Sériová verzia bola implementovaná pomocou vstavaných metód triedy `ndarray` z knižnice numpy [15]. Paralelná verzia bola implementovaná pomocou CUDA kernelu spusteného z programu jazyku Python. Počet FLOP v rámci jednej operácie SAXPY je vypočítaný ako počet prvkov vektoru vynásobený konštantou 2, keďže na výpočet každého prvku výsledného vektoru je potrebná jedna operácia násobenia a jedna operácia sčítania.

Sériová verzia dosahuje maximálny výpočtový výkon približne 3100 MFLOPS pri vektore o cca. 16-tisíc prvkoch (2^{14}). Pri väčších veľkostiach vektoru výkon sériovej verzie klesá. Príčinou je pravdepodobne nedostatok vyrovnávacej pamäte a tým spôsobená nutnosť načítať dáta z hlavnej

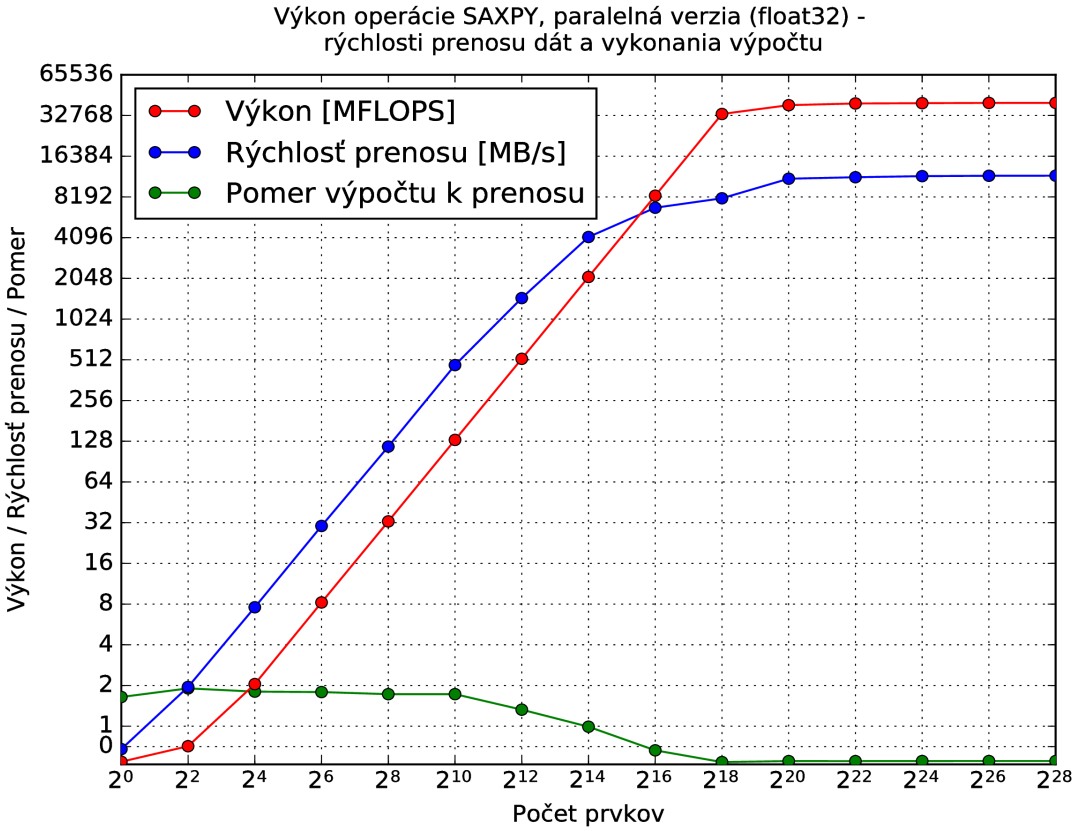
pamäte, čo operáciu spomaľuje. Pri veľkosti vektoru od 1 do 8192 (2^{13}) prvkov je sériová verzia priemerne 5,6-krát výkonnejšia než paralelná verzia. Je to spôsobené vysokou cenou prenosu dát medzi pamäťou CPU a GPU a spustením kernelu na GPU.



Graf 3.3: Porovnanie výkonu SAXPY použitím knižnice numpy (metódy `__add__` a `__mul__` triedy `ndarray`) a PyCUDA + CUDA kernelu, vzhľadom k veľkosti vektoru. Jedna operácia SAXPY nad vektormi o veľkosti *n* zodpovedá *2n* FLOP (1x násobenie, 1x sčítanie).

Paralelná verzia dosahuje vyšší výkon až pri vektoroch o veľkosti cca. 32-tisíc (2^{15}) prvkov a viac. Priemerne je jej výkon 22,4-krát vyšší než u sériovej verzie pre vektory o veľkosti 64-tisíc (2^{16}) až 256 miliónov (2^{28}) prvkov. Nárast výkonu vzhľadom k veľkosti vektoru je lineárny až po veľkosť cca. 256-tisíc (2^{18}) prvkov, pri ktorej je obmedzujúcim faktorom rýchlosť prenosu dát medzi pamäťou CPU a GPU (viď graf 3.4). Objem prenášaných dát je pri väčších vektoroch relatívne veľký, vzhľadom k nízkej výpočtovej náročnosti operácie vykonávanej nad dátami. To má za následok zníženie prírastku výkonu s rastúcou veľkosťou vektoru. Toto zníženie prírastku výkonu je badateľné od momentu, keď začne byť prenos dát časovo náročnejší než samotný výpočet.

Graf 3.4 je špecifický pre paralelnú verziu. Ukazuje rýchlosť prenosu dát, výkon kernelu a pomer časovej náročnosti výpočtu k časovej náročnosti prenosu dát vzhľadom k počtu prvkov vstupných vektorov. Zníženie pomeru trvania výpočtu k trvaniu prenosu má za následok spomínané zníženie prírastku výkonu s rastúcou veľkosťou vektoru. Pri vektore o 256-tisíc (2^{18}) prvkoch začína byť na prenos dát vynaložený výrazne dlhší čas než na výpočet – pomer klesá výrazne pod hodnotu 1. Táto veľkosť vektoru korešponduje s veľkosťou, pri ktorej prestáva rásť výkon paralelnej verzie.

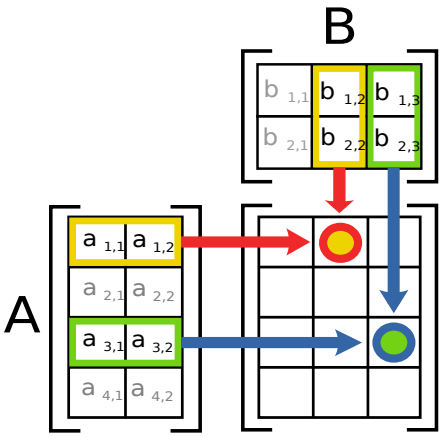


Graf 3.4: Paralelná verzia SAXPY – výkon kernelu (do meraného času nie je započítaný čas prenosu dát), rýchlosť prenosu dát (kombinovaná rýchlosť z CPU na GPU a opačným smerom), a pomer času potrebného na vykonanie kernelu k času potrebnému na prenos dát, vzhľadom k veľkosti vektoru.

3.3 Násobenie matíc

Násobenie matíc je binárna operácia. Jej vstupom sú matice A o rozmeroch $m \times n$ a B o rozmeroch $n \times p$. Výstupom je matica AB o rozmeroch $m \times p$. Prvok v i-tom riadku a j-tom stĺpci výslednej matice AB je rovný skalárnemu súčinu vektoru i-tého riadku matice A s vektorom j-tého stĺpca matice B. Pre ilustráciu viď obrázok 3.1. V tomto teste boli použité iba štvorcové matice, a teda všetky matice (vstupné aj výstupná) majú rovnaké rozmery. Asymptotická výpočtová náročnosť tejto operácie je $O(n^3)$ pre štvorcovú maticu o strane n prvkov. Počet výsledných prvkov je $n \times n$ a pre každý prvok výslednej matice je potrebných n operácií násobenia a $n-1$ operácií sčítania.

Graf 3.5 znázorňuje výkon sériovej a paralelnej verzie vzhľadom k veľkosti strany matice. Sériová verzia je implementovaná pomocou metódy `dot` triedy `ndarray` z knižnice numpy. Paralelná verzia je implementovaná pomocou CUDA kernelu spusteného z programov jazyku Python a C++. Pre porovnanie je v grafe zahrnutý tiež výkon paralelnej verzie v jazyku Python zahrňujúci prenos dát. Pri výpočte tohto výkonu bol počet operácií (FLOP) delený celkovým časom, teda časom zahrňujúcim okrem vlastného výpočtu tiež prenos vstupných a výstupných dát. Výkon násobenia matíc A $(m \times n)$ a B $(n \times p)$ je vypočítaný ako $m \times p \times (2n - 1)$, keďže skalárny súčin riadku matice A a stĺpca matice B vyžaduje n násobení a $n - 1$ sčítaní, a počet vykonaných skalárnych násobení vektorov je $m \times p$.



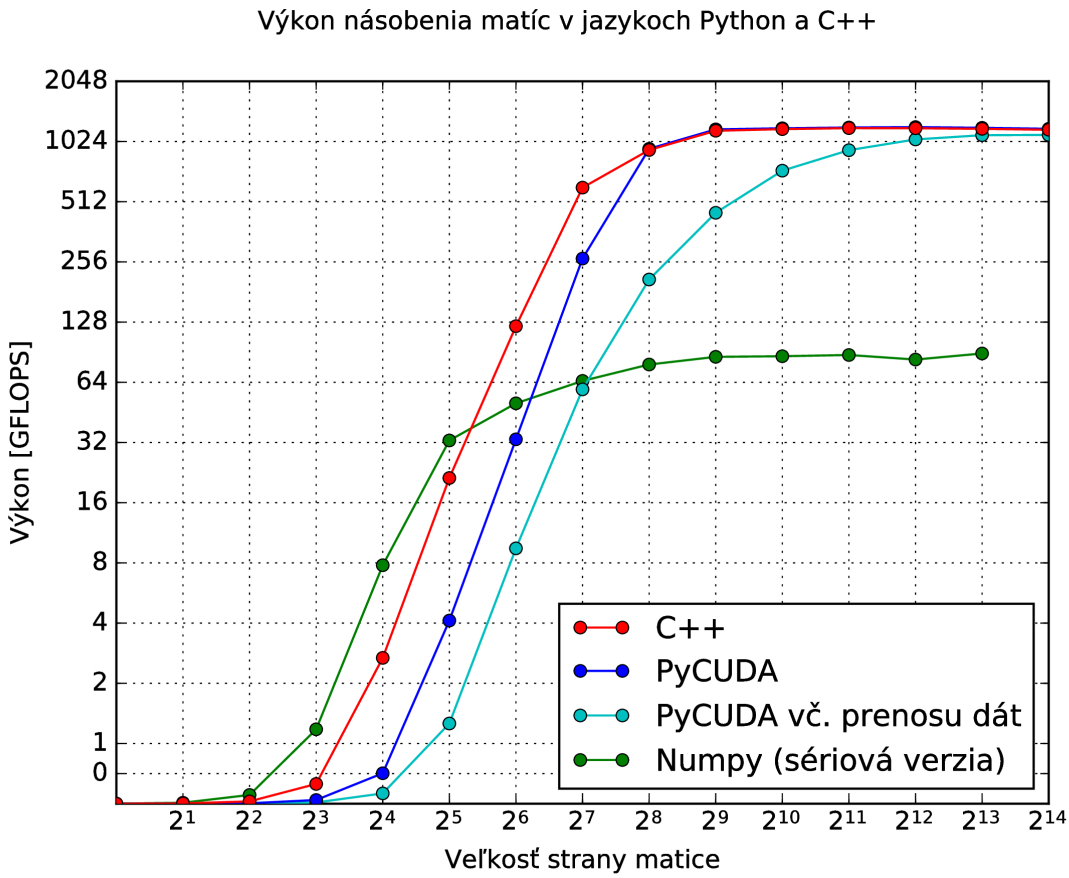
Obrázok 3.1: Násobenie matíc. Krúžok vo výslednej matici znázorňuje skalárny súčin vektorov. [24]

Výkon sériovej verzie je vyšší než výkon ostatných verzií pri maticiach o strane 32 (2^5) a menej prvkov. Pri väčších maticiach je teoreticky výkonnejšia paralelná verzia. Ako ale vidno z grafu 3.5, celkové trvanie výpočtu vrátane času potrebného na prenos dát je u paralelnej verzie nižšie až pri veľkostiach strany nad 128 (2^7) prvkov. Pri týchto veľkostiach je výkon paralelnej verzie v jazyku Python v priemere 13,5-krát vyšší (9,3-krát pri započítaní času prenosu dát). Pri veľkostiach strany 128 (2^7) a menej je výkon sériovej verzie priemerne 2,2-krát vyšší než výkon paralelnej verzie v jazyku Python so započítaním času prenosu dát. Maximálny výkon sériovej verzie okolo 80-90 GFLOPS je dosiahnutý pri maticiach o strane 256 (2^8) prvkov a viac. Pri tejto hodnote je zrejme dosiahnutý tiež maximálny možný výkon procesoru pre danú úlohu. Obmedzujúcim faktorom je teda výkon procesoru, na rozdiel od úlohy SAXPY z kap. 3.2, kde týmto faktorom bola rýchlosť hlavnej pamäte, resp. veľkosť vyrovnávacej pamäte.

Paralelná verzia dosahuje maximálny výkon cca. 1200 GFLOPS, pričom pri maticiach o strane 512 (2^9) prvkov a viac je výkon takmer konštantný. Toto je zrejme spôsobené relatívne pomalou hlavnou pamäťou GPU – prístup do hlavnej pamäti GPU spomaľuje výpočet. Napriek použitiu rýchlejšej zdieľanej pamäte je z teoretického maximálneho výkonu takmer 9 TFLOPS [25] využitá iba malá časť. Výkon by bolo možné zvýšiť napríklad použitím optimalizovaného kernelu, ktorý by lepšie využíval dostupné prostriedky. Napríklad pre zlepšenie prístupu do pamäti by mohla jedna z matíc byť pred výpočtom transponovaná.

U verzie Python je výkon kernelu v priemere o 3,98 % nižší než u verzie C++. Tieto verzie sa značne líšia v náročnosti implementácie. Dokazuje to aj rozsah zdrojového kódu – počet riadkov zdrojového kódu je 350 u verzie C++ a 176 u verzie Python.

Výkon vypočítaný z čistého času výpočtu sa blíži výkonu vypočítanému z celkového času (vrátane času prenosu dát). Z toho je vidno, že násobenie matíc je výpočtovo náročná operácia, na rozdiel od SAXPY. Je preto vhodnejšia pre akceleráciu na GPU. Dokazuje to zvyšujúci sa podiel trvania výpočtu na celkovom čase, kým u SAXPY sa tento podiel znižoval (viď graf 3.4, „Pomer výpočtu k prenosu“). Pri dostatočne veľkých maticiach je rozdiel medzi čistým časom výpočtu a celkovým časom relatívne malý, v porovnaní s menšími veľkosťami matíc.



Graf 3.5: Porovnanie výkonu násobenia štvorcových matíc použitím knižnice numpy (metódy `__add__` a `__mul__` triedy `ndarray`) a CUDA kernelu v jazykoch Python (PyCUDA) a C++, vzhľadom k veľkosti strany matice. Počet FLOP pri násobení matíc *A (m×n)* a *B (n×p)* je rovný $m \times p \times (2n - 1)$.

4 Súčasný stav

Táto kapitola obsahuje stručné zhrnutie práce v prvom semestri a plán pokračovania práce v druhom semestri.

Vykonaná práca v prvom semestri:

- Návrh a implementácia mikrotestov, ktoré slúžili na oboznámenie sa s platformou CUDA a knižnicou PyCUDA.
- Prevedenie meraní pomocou mikrotestov na školskom GPU serveri. Počítač Anselm bol využívaný minimálne. Dôvodom je jednoduchší prístup a modernejšie hardvérové vybavenie školského serveru (výkonnejšie GPU).
- Z testov boli vyvodené závery:
  - pamäť je dobré alokovať ako „pinned“,
  - rýchlosť GPU pamäte je často obmedzujúcim faktorom,
  - čas potrebný na prenos dát je u výpočtovo nenáročných úloh (napr. operácia SAXPY, násobenie malých matíc apod.) nezanedbateľný,
  - na GPU sa preto oplatí akcelerovať hlavne veľké, výpočtovo náročné úlohy.
- Mikrotesty a ich výsledky sú popísané v kapitole 3.

Práca v druhom semestri:

- Dr. Jaroš poskytol C++ kód implementujúci riešenie problému ruksaku (*knapsack problem*) pomocou genetického algoritmu. Konkrétne sa jedná o jednodimenzionálny 0/1 problém ruksaku.
  - Existujúci kód obsahuje CUDA kód vykonávajúci výpočet a obslužný kód napísaný v C++.
  - Funkcionalita CUDA kódu je členená do niekoľkých kernelov.
  - Funkcionalita obslužného kódu je vhodne členená do tried implementovaných v oddelených súboroch.
  - Kód je dostatočne komentovaný, i keď neobsahuje dokumentáciu.
  - Obsahuje meranie času, ktoré je však vykonané na CPU, nie na GPU.
- Obslužný kód má byť prepísaný z C++ do Pythonu, pričom kernely budú zachované.
  - Python kód bude vyvíjaný pomocou systému Git [26].
  - Súčasný C++ kód obsahuje niekoľko elementov, ktoré môžu spôsobiť problémy pri prepisovaní do Pythonu:
    - tieto problémy budú musieť byť analyzované a vhodne vyriešené,
    - problémy budú dokumentované v systéme Git ako *issues*.
  - Výsledný Python kód bude vybavený dokumentáciou.
  - Návrh prepisu kódu do jazyka Python je obsiahnutý v kapitole 5.
  - Detaily implementácie sú popísané v kapitole 6.

- Má byť porovnaná funkčnosť a výkonnosť C++ a Python verzie.
  - Plánované merania:
    - testovanie na školskom serveri SC-GPU1 a superpočítači Anselm,
    - porovnanie funkčnosti použitím pevne daného semienka pre generátor pseudonáhodných čísel (ďalej „náhodné semienko“),
    - porovnanie výkonnosti kernelu (mala by byť podobná u oboch verzií) a obslužného kódu.
  - Existujúci C++ kód nie je dostatočne prispôsobený na plánované testovanie. Bude potrebné do kódu pridať:
    - spoľahlivé meranie času na GPU s využitím CUDA events [16],
    - možnosť použitia pevne daného náhodného semienka.
  - Výsledky testovania sú popísané v kapitole 7.

5 Návrh riešenia problému ruksaku

V tejto kapitole je rozobraný existujúci obslužný kód napísaný v jazyku C++. Sú popísané problémy, ktoré môžu nastať počas jeho prepisovania do jazyku Python. Tie sú rozobrané v jednotlivých podkapitolách. Sú tiež popísané zmeny, ktoré je potrebné vykonať na C++ kóde za účelom porovnania výkonu verzií C++ a Python.

Tieto problémy bude potrebné vhodným spôsobom vyriešiť. Návrh možných riešení problémov, ktoré boli odhalené pri študovaní C++ kódu, je obsiahnutý v tejto kapitole. Konkrétne riešenia týchto problémov sú potom popísané v kapitole 6 spolu s príkladmi zdrojových kódov.

Pre podrobný návrh riešenia a jeho správnu implementáciu bude potrebné dôkladne naštudovať existujúci kód v jazyku C++. Počas prepisovania tohto kódu do jazyku Python môžu nastať ďalšie komplikácie. Takéto komplikácie, ktoré neboli odhalené počas prvotného návrhu popísaného v tejto kapitole, budú taktiež popísané v kapitole 6.

5.1 Obecný postup

Existujúci obslužný kód v C++ je implementovaný objektovo-orientovaným spôsobom. Logika kódu je rozdelená do niekoľkých tried definovaných v samostatných súboroch. Jednotlivé súbory väčšinou obsahujú definíciu jednej triedy, prípadne pomocných prvkov ako sú napríklad konštanty alebo definície typov. Výnimku tvoria súbory `main.cpp`, `CUDA_Kernels.cu` a `CUDA_Kernels.h`.

Súbor `main.cpp` obsahuje funkciu `main`, ktorá tvorí vstupný bod programu. Tento vstupný bod bude pri prepisovaní potrebné pozmeniť na jeho ekvivalent v jazyku Python. Funkcia `main` môže byť preložená z C++ v jej súčasnej podobe, ale je potrebné ju explicitne zavolať v prípade, že je splnená podmienka `__name__ == '__main__'` [27].

Súbory `CUDA_Kernels.cu` a `CUDA_Kernels.h` obsahujú CUDA kernely, ako značí ich názov. Keďže do jazyku Python bude prekladaný iba obslužný kód, nie kernely, tieto súbory by mali byť zachované v pôvodnej podobe. Môže však byť nutné vykonať určité menšie zmeny.

Všetky ostatné súbory, obsahujúce implementácie tried, budú prepísané do jazyka Python. Pri tomto prepisovaní bude zachovaná funkčnosť a štruktúra pôvodného kódu. To znamená že Python kód bude obsahovať triedy z pôvodného C++ kódu a tieto triedy budú poskytovať rovnakú funkcionalitu ako pôvodné C++ triedy. Členenie kódu do zdrojových súborov podľa tried by tiež malo ostať zachované. Líšiť sa bude konkrétna implementácia, kde pri prepisovaní budú konštrukcie jazyka C++ nahradené ich vhodným ekvivalentom v jazyku Python. Budú tiež identifikované také konštrukcie a úseky kódu, ktoré je možné v jazyku Python zapísať jednoduchšie či idiomatickejšie, a tie budú vhodným spôsobom prepísané.

Výsledkom budú zdrojové súbory jazyku Python implementujúce obslužný kód. Ich štruktúra by mala byť približne rovnaká ako štruktúra C++ kódu. To platí ako pre fyzickú štruktúru (členenie zdrojového kódu do súborov), tak pre logickú štruktúru (členenie funkcionality do tried, metód, atď). K zdrojovým súborom jazyka Python budú priložené súbory `CUDA_Kernels.cu` a `CUDA_Kernels.h` obsahujúce zdrojový kód kernelov.

Preložený zdrojový kód bude podporovať spustenie pomocou interpretu jazyka Python verzie 3. V prípade potreby môže byť kód upravený, aby bol kompatibilný s jazykom Python verzie 2.

5.2 Triedy obsahujúce štruktúry

V obslužnom zdrojovom kóde je logika genetického algoritmu rozdelená do tried. Tieto triedy udržiavajú vstupné a výstupné dáta algoritmu na CPU aj na GPU a iné pomocné údaje. Tiež obsahujú obslužné metódy volané z CPU kódu. Tieto metódy slúžia napríklad na spustenie výpočtu na GPU, prenos dát medzi CPU a GPU apod. Pomocné metódy sú spúšťané na procesore a kernely implementujúce samotný genetický algoritmus sú spúšťané na GPU. Z GPU nie je možné volať pomocné metódy a ich spustenie na GPU by ani nebolo zmysluplné, keďže sa tieto metódy netýkajú samotného výpočtu, ale iba obslužných úkonov. Je teda nežiaduce, aby boli medzi CPU a GPU prenášané celé inštancie týchto tried obsahujúce nepotrebné dáta a ukazovatele na metódy. Treba preniesť iba dáta potrebné k výpočtu, čím sa tiež zvýši efektivita komunikácie medzi CPU a GPU.

Nastáva teda problém, ako efektívne komunikovať relevantné vstupné a výstupné dáta medzi CPU a GPU. Zároveň musia na CPU byť dostupné obslužné metódy, ktoré na GPU spustiteľné nebudú. Inak povedané, je potrebné oddeliť dáta algoritmu od pomocnej funkcionality.

V súčasnom C++ kóde je tento problém vyriešený použitím štruktúr. Napríklad trieda `TGlobalKnapsackData` obsahuje ukazovatele na štruktúru `TKnapsackData` v pamäti CPU a v pamäti GPU, a tiež pomocné metódy umožňujúce prenos dát medzi CPU a GPU. Štruktúra `TKnapsackData` potom obsahuje samotné dáta výpočtu, ako napríklad počet predmetov, nosnosť ruksaku a ukazovatele na polia váh a cien predmetov. Pred výpočtom sa načítajú vstupné dáta zo súboru a zapíšu sa do štruktúry uloženej v pamäti CPU. Tá je potom prenesená do pamäti GPU, kde ku nej pristupujú kernely vykonávajúce výpočet. Počas výpočtu sú do pamäte CPU priebežne prenášané štatistické údaje, ktoré sú zobrazované na konzolový výstup. Po ukončení výpočtu sú výsledky prenesené na CPU a zobrazené na výstup, podobne ako priebežné štatistiky.

Všetok kód spúšťaný z CPU bude prepísaný do jazyku Python, menovite spomínané triedy obsahujúce štruktúry. Vzniká problém, ako reprezentovať v jazyku Python štruktúry jazyka C++. V ideálnom prípade by mala byť zvolená taká reprezentácia, aby súčasná podoba kernelov zostala zachovaná. To znamená, že binárna podoba dát prenášaných na GPU musí ostať nezmenená. Zvolená reprezentácia štruktúr v jazyku Python musí mať rovnakú binárnu podobu, ako pôvodné štruktúry v jazyku C++. Zároveň musia byť dáta uložené v štruktúrach prístupné z jazyku Python. Musí byť totiž možné ich napríklad načítať zo súboru alebo zobraziť na výstup, a tieto operácie budú vykonávane v jazyku Python.

V princípe je teda potrebné nejakým spôsobom previesť premenné jazyku Python do binárnej podoby akceptovanej kernelmi. Typy týchto premenných budú vstavané typy jazyku Python, napríklad `int` alebo `float`. Samozrejme je tiež potrebný opačný prevod – z binárnej podoby na premennú vstavaného typu. Ďalej je potrebné určitým spôsobom premenné v binárnej podobe konsolidovať do štruktúry, respektíve z tejto štruktúry získať jednotlivé binárne hodnoty.

Jedným z riešení je použitie knižnice struct [28]. Táto vstavaná knižnica jazyka Python je určená práve na tento účel. Umožňuje prevádzať typy jazyka Python na typy kompatibilné s jazykom C/C++. Taktiež umožňuje vytvoriť binárnu reprezentáciu štruktúry obsahujúcu prevedené hodnoty.

Na vytvorenie binárnej reprezentácie slúži funkcia `struct.pack`. Táto funkcia dostane ako argument formátovací reťazec, na základe ktorého vytvorí binárnu reprezentáciu hodnôt. Samotné hodnoty sú predané funkcii ako ostatné argumenty za formátovacím reťazcom. Tieto hodnoty sú štandardné hodnoty jazyka Python, ich typy môžu byť napríklad `int` alebo `float`. Formátovací reťazec obsahuje informácie o tom, na aké C-typy majú byť vstupné hodnoty prevedené. Napríklad formátovací príkaz `b` reprezentuje C-typ `char`, teda 8-bitovú celočíselnú hodnotu so znamienkom. To znamená, že typ príslušného argumentu musí byť `int` alebo `long`, keďže to sú jediné celočíselné typy v jazyku Python. Jeho hodnota musí byť v rozsahu [-128; 127], čo zodpovedá rozsahu 8-bitového celého čísla so znamienkom v dvojkovom doplnku.

Funkcia `struct.pack` vracia hodnotu typu `bytes` (vo verzii Python 3) respektíve `str` (Python 2). Táto hodnota obsahuje vstupné hodnoty zakódované podľa formátovacieho reťazca. Hodnoty typu `bytes` a `str` však nie sú modifikovateľné, tzn. jednotlivé položky štruktúry nie je možné meniť. Pri prenose z GPU späť na CPU je ale potrebné do štruktúry zapísať dáta z GPU, teda je potrebné, aby štruktúra bola modifikovateľná. Preto je potrebné hodnotu previesť na typ `bytearray`, ktorý je modifikovateľný. Kód 5.1 obsahuje príklad použitia funkcie `struct.pack`.

```
>>> struct.pack('bif', 35, -700, 2.75)
b'#\x00\x00\x00D\xfd\xff\xff\x00\x000@'
```

Kód 5.1: Použitie funkcie `struct.pack`. Prvá vstupná hodnota (`35`) je typu `byte`, druhá (`-700`) je typu `int`, tretia (`2,75`) je typu `float`. Prvý znak výslednej hodnoty tvorí prvá hodnota, za ním nasledujú tri nulové znaky pre zarovnanie na šírku slova. Druhá hodnota tvorí znaky 5-8, tretia hodnota znaky 9-12.

Funkcia `struct.unpack` slúži na spätnú konverziu z binárnej reprezentácie na vstavané typy jazyka Python. Ako argument dostane formátovací reťazec a binárnu hodnotu. Táto hodnota môže byť napríklad tá, ktorá bola získaná funkciou `struct.pack`, alebo tiež hodnota typu `bytearray`. Kód 5.2 obsahuje príklad použitia funkcie `struct.unpack`.

```
>>> struct.unpack('bif',
                  b'#\x00\x00\x00D\xfd\xff\xff\x00\x000@')
(35, -700, 2.75)
```

Kód 5.2: Použitie funkcie `struct.unpack`. Pre popis vstupných a výstupných hodnôt viď kód 5.1.

Prenos dát medzi CPU a GPU bude následne realizovaný pomocou funkcie `pycuda.driver.memcpy_htod`, ktorej bude predaná hodnota typu `bytearray`. To znázorňuje kód 5.3.

```
>>> cpu_struct = bytearray(struct.pack('bif', 35, -700, 2.75))
>>> pycuda.driver.memcpy_htod(gpu_struct, cpu_struct)
```

Kód 5.3: Prenos dát vytvorených pomocou funkcie `struct.pack` na GPU. Premenná `cpu_struct` je modifikovateľná hodnota uložená v pamäti CPU, ktorá je prenášaná na GPU.

5.3 Definície štruktúr v hlavičkových súboroch

Pôvodný C++ kód je rozdelený do niekoľkých hlavičkových a implementačných súborov. Každá dvojica súborov tvorí určitý logický celok podieľajúci sa na celkovej funkcionalite. V hlavičkových súboroch sú definované dátové štruktúry, ktoré udržiavajú dáta prenášané medzi CPU a GPU. Ukazovatele na tieto štruktúry sú potom predávané kernelom. Kernely samotné sú definované v súboroch `CUDA_Kernels.h` a `CUDA_Kernels.cu`. To, že kernely používajú spomínané štruktúry znamená, že súbory obsahujúce kernely sú závislé na súboroch definujúcich štruktúry.

Podľa zadania má byť všetok C++ kód okrem kernelov prepísaný do Pythonu. Výsledný C++ kód volaný z Pythonu by mal byť teda tvorený iba súbormi `CUDA_Kernels.h` a `CUDA_Kernels.cu`. Tu vzniká problém – nemožno odstrániť hlavičkové súbory, pretože kernely sú na nich závislé.

Existuje niekoľko možných riešení. Jedno riešenie je, že existujúce hlavičkové súbory budú zachované a priložené k súborom obsahujúcim kernely. Problém tohto riešenia je, že hlavičkové súbory obsahujú okrem definícií štruktúr tiež definície tried. Tie zahrňujú príslušné deklarácie obslužných metód spúšťaných na CPU. Tieto nadbytočné deklarácie nie sú pre fungovanie kernelov potrebné a bolo by teda vhodné ich odstrániť. Potom ale budú jednotlivé hlavičkové súbory pozostávať iba z definície jednej, relatívne malej, štruktúry. Ich objem bude zrejme dostatočne malý na to, aby bolo možné ich spojiť do jedného súboru. Navyše nebudú používané z nijakého iného súboru než `CUDA_Kernels.cu`, takže toto spojenie neovplyvní iné existujúce súbory.

Druhým riešením, ako už bolo naznačené, je spojiť všetky potrebné hlavičkové súbory do jedného. Týmto súborom by mohol byť samotný súbor `CUDA_Kernels.h`. Z ostatných súborov budú vyňaté iba tie definície, ktoré sú skutočne potrebné pre preklad kernelov. Tieto definície budú presunuté do súboru `CUDA_Kernels.h`. Implementačný súbor `CUDA_Kernels.cu` bude tak závislý iba na svojom vlastnom hlavičkovom súbore.

5.4 Globálny zámok na GPU

V rámci genetického algoritmu bežiaceho na GPU sa okrem samotného procesu evolúcie vykonáva tiež výpočet štatistík súčasnej populácie, a to za užívateľom zadaný počet generácií. Tieto štatistiky sú potom prenesené na CPU a zobrazené užívateľovi. Štatistiky obsahujú údaje napríklad o priemernej hodnote jedinca vypočítanej ohodnocovacou funkciou.

Pri výpočte priemernej hodnoty jedinca je potrebné sčítať hodnoty všetkých jedincov. Tento výpočet prebieha v kerneli `CalculateStatistics` a pozostáva z niekoľkých krokov. Najprv každé vlákno v bloku sčíta jednu alebo viac hodnôt do zdieľanej pamäte bloku. Potom sú hodnoty v zdieľanej pamäti každého bloku postupne sčítané po dvojiciach s postupne sa zmenšujúcim rozostupom (tzv. redukcia). Nakoniec sú čiastočné súčty jednotlivých blokov sčítané do globálnej pamäte. Na tento posledný krok je potrebné zabezpečiť výlučný prístup, keďže všetky bloky pristupujú k tej istej adrese v globálnej pamäti. Je potrebný globálny zámok na GPU, ktorý synchronizuje všetky bloky.

Tento zámok je v C++ kóde implementovaný ako kompozitná trieda `TGPU_Lock`. Pojem „kompozitná“ značí, že časť kódu tejto triedy je vykonávaná na CPU a časť na GPU. Konkrétne je táto trieda inštanciovaná z CPU, ale samotné zamykanie a odomykanie je vykonávané na GPU. V konštruktore triedy, vykonanom na CPU, sa alokuje v pamäti GPU premenná typu `int` a tá sa inicializuje na hodnotu 0. Ukazovateľ na túto hodnotu je v inštancii triedy uložený pod názvom `mutex`. Na zamykanie a odomykanie slúžia metódy `Lock` a `Unlock`, spúšťané na GPU. Tieto metódy využívajú CUDA funkcie `atomicCAS` na zamykanie a `atomicExch` na odomykanie zámku. Funkcie pracujú s ukazovateľom na hodnotu typu `int`, ktorým je v tomto prípade inštančná premenná `mutex`.

Kernel `CalculateStatistics` je spúšťaný z metódy `Calculate` triedy `TGPU_Statistics`. Táto metóda je spúšťaná na CPU a je v nej vytvorená inštancia `TGPU_Lock`. Potom je z nej volaný kernel, ktorému je táto inštancia `TGPU_Lock` predaná ako argument. Nad ňou sú potom v kerneli volané metódy `Lock` a `Unlock`.

Problém spočíva v tom, že kernelu `CalculateStatistics` je predávaná inštancia triedy `TGPU_Lock` a táto trieda má byť prepísaná do jazyka Python. Ak by trieda bola prepísaná do jazyka Python, potom by ju nebolo možné predať ako argument kernelu, ktorý je napísaný v CUDA C++. Ak by ale bola ponechaná implementácia tejto triedy v CUDA C++, tak by ju nebolo možné použiť z jazyka Python.

Evidentne bude potrebné rozdeliť triedu `TGPU_Lock` na CPU-časť, ktorá bude prepísaná do Pythonu, a GPU-časť, ktorá ostane napísaná v C++. Jeden možný prístup je vyčleniť z triedy metódy vykonávané na GPU, a tieto metódy prepísať na funkcie. Na GPU sú vykonávané metódy `Lock` a `Unlock`, ktoré pracujú iba s inštančnou premennou `mutex`. To znamená, že môžu byť prepísané na funkcie, ktoré dostanú ako argument priamo onen ukazovateľ, ktorý premenná `mutex` reprezentuje. CPU-časť potom iba alokuje hodnotu typu `int` v pamäti GPU a pri volaní kernelu `CalculateStatistics` mu predá ukazovateľ túto hodnotu. V kerneli bude ukazovateľ ďalej predaný funkciám `Lock` a `Unlock`.

5.5 Singleton triedy

Pri spustení programu môže užívateľ zadať parametre genetického algoritmu, ako napríklad veľkosť populácie alebo počet generácií. Väčšina parametrov je nepovinných a má predvolené hodnoty. Jediným povinným parametrom je relatívna cesta k súboru so vstupnými dátami, ktorý obsahuje váhy a ceny predmetov a nosnosť ruksaku. Parametre algoritmu sú používané na niekoľkých rôznych miestach v programe, pričom v niektorých triedach sú napríklad pozmenené alebo pridané určité údaje, parametre sú prenášané na GPU, atď. Je teda potrebné, aby k nim všetky tieto triedy mali prístup a boli schopné ich modifikovať. Tieto modifikácie sa musia prejaviť vo všetkých ostatných miestach, kde sú parametre použité. Dá sa povedať, že parametre sú v tomto programe určitou formou globálnych dát. Nie je však vhodné ich implementovať ako skutočné globálne premenné.

Na uchovanie parametrov je v C++ kóde použitá singleton [29] trieda `TParameters`. Statická metóda `GetInstance` vracia ukazovateľ na singleton inštanciu. Singleton inštancia je uložená v triede ako statická premenná `pTParametersSingle`. Pokiaľ inštancia ešte neexistuje, tak ju metóda `GetInstance` vytvorí a uloží do premennej `pTParametersSingle`.

Jazyk Python neobsahuje syntaktický ekvivalent kľúčového slova `static` jazyku C++. Inštančné premenné sú v jazyku Python definované priamo v konštruktore, bez deklarácie v tele triedy, ako je tomu u C++. Definície premenných v tele triedy sú však možné a takéto definície definujú triedne premenné. Pojem triedna premenná v jazyku Python je ekvivalentný s pojmom statická premenná v jazyku C++. Oba pojmy označujú atribút triedy. Obdobne ekvivalentom statickej metódy v C++ je triedna metóda. Triedne metódy sú v jazyku Python definované pomocou dekorátoru `classmethod`. Pri volaní potom dostanú ako argument triedu namiesto inštancie. Existuje tiež podobný dekorátor `staticmethod`, ktorý by sa na prvý pohľad mohol zdať vhodnejší, keďže sa jeho názov ponáša na kľúčové slovo `static`. Táto podobnosť je však zavádzajúca – statická metóda v jazyku Python pri zavolaní nedostane ako argument ani inštanciu, ani triedu. Dostane iba užívateľom zadané argumenty.

Naskytá sa otázka, či ponechať súčasný singleton model s triednou premennou a triednou metódou, alebo uložiť parametre priamo do triedy ako triedne premenné. Tým by odpadla nutnosť vytvárať, udržiavať a volaním metódy získavať singleton inštanciu. Namiesto singleton inštancie by sa používala priamo singleton trieda.

5.6 Preklad a spúšťanie kernelov

Jazyk CUDA C++ umožňuje volať kernely podobne, ako sú volané štandardné funkcie jazyka C++. Obslužný kód a kód kernelov nie sú od seba oddelené. Môžu byť napríklad spolu v tom istom súbore, alebo dokonca existovať spoločne v rámci jednej triedy, ako je tomu v prípade `TGPU_Lock`

(viď kapitola 5.4). Obslužný kód v C++ je prekladaný spolu s kernelmi prekladačom `nvcc` (*Nvidia CUDA Compiler*) [30]. Výsledkom prekladu je binárny súbor obsahujúci ako obslužný kód, tak kód kernelov.

Z jazyka Python je takisto možné volať kernely ako štandardné funkcie, lenže predtým musia byť kernely preložené. Na tento preklad slúži prekladač `nvcc`, ktorý je za behu programu volaný z knižnice PyCUDA použitím triedy `pycuda.compiler.SourceModule`. Argumentom konštruktoru tejto triedy je zdrojový kód, tak ako by bol obsiahnutý v zdrojovom súbore CUDA C++. V konštruktore je pomocou `nvcc` tento kód preložený. Výsledkom prekladu je tzv. PyCUDA modul, reprezentovaný inštanciou triedy `pycuda.compiler.SourceModule`. Symboly definované v zdrojovom kóde, ako napríklad kernely alebo konštanty, sú dostupné prostredníctvom inštancie modulu. Kód 5.4 znázorňuje vytvorenie modulu a získanie kernelu s názvom `CalculateStatistics`.

```
module = pycuda.compiler.SourceModule(source_text)
stats_kernel = module.get_function("CalculateStatistics")
```

Kód 5.4: Preklad zdrojového kódu do `pycuda.compiler.SourceModule` a získanie funkcie kernelu. Argument `source_text` je textový reťazec obsahujúci zdrojový kód v CUDA C++. Premenná `stats_kernel` je funkcia predstavujúca CUDA kernel a je možné ju volať ako štandardnú funkciu.

Kernely je potrebné preložiť iba raz, ideálne pri spustení programu. Preložením zdrojového súboru `CUDA_Kernels.cu`, obsahujúceho všetky kernely, vznikne PyCUDA modul. Tento modul musí byť globálne dostupný všetkému obslužnému kódu. Otázkou je, ako zabezpečiť preklad kernelov a ich sprístupnenie obslužnému kódu.

Riešením je použitie návrhového vzoru singleton [29]. V tomto konkrétnom prípade dôjde pri vytvorení singleton inštancie k prekladu zdrojového súboru `CUDA_Kernels.cu`. Výsledný preložený modul bude uložený v singleton inštancii, prostredníctvom ktorej bude sprístupnený celému obslužnému kódu.

5.7 Meranie trvania výpočtu

Cieľom tejto práce je porovnať výkon implementácií v jazykoch C++ a Python. Pre porovnanie výkonu je potrebné presné meranie času. Kód oboch verzií pozostáva z obslužného kódu a CUDA kernelov. CUDA kernely by mali ostať zachované a prípadné nutné zmeny by mali mať minimálny vplyv na ich fungovanie a predovšetkým výkon.

V súčasnom C++ kóde je implementované rudimentárne meranie trvania behu kernelov. Je pri ňom použitá funkcia `clock`, ktorá je volaná tesne pred spustením genetického algoritmu na GPU. Meraný čas nezahrňuje inicializačnú fázu, ktorú tvorí napríklad načítanie programu do pamäte systému či spracovanie argumentov programu. Funkcia `clock` vracia počet taktov procesoru spotrebovaných volajúcim programom. Tento počet taktov je potom prevedený na čas v sekundách delením jeho hodnoty konštantou `CLOCKS_PER_SEC`. Kernely sú ale spúšťané na GPU, teda trvanie ich behu by malo byť merané hardvérom GPU, nie procesorom. Súčasná implementácia nemeria spoľahlivo ani trvanie samotného genetického algoritmu na GPU, ani celkové trvanie behu programu vrátane inicializácie. Spôsob merania času v súčasnej implementácii bude musieť byť zmenený.

Predpokladá sa, že výkon kernelov bude vo verziách C++ a Python približne rovnaký. Naopak výkon obslužného kódu bude u verzie Python s najvyššou pravdepodobnosťou nižší, než u verzie C++. Prvý predpoklad bude potrebné overiť meraním výkonu kernelov. Je dôležité, aby bol

čo najpresnejšie zmeraný čas potrebný na vykonanie algoritmu na GPU, pričom do tohto merania nesmie byť zahrnuté vykonanie obslužného kódu. Potom je potrebné porovnať časovú náročnosť obslužného kódu verzií C++ a Python.

Výkon kernelov bude spoľahlivo meraný použitím CUDA events [16]. Tento postup bol použitý v tiež v mikrotestoch. To bude vyžadovať zásah do existujúceho C++ kódu, aby bol doplnený o potrebnú funkcionalitu. Kernely však ostanú nezmenené.

Meranie behu celého programu môže byť založené napríklad na súčasnej implementácii, pričom bude potrebné do merania zahrnúť tiež inicializačnú fázu programu. V súčasnosti sa meranie začína po inicializačnej fáze a zahrňuje iba beh algoritmu na GPU. To znamená, že zahájenie merania času bude treba presunúť na začiatok programu. Podobné meranie bude implementované v Python verzii.

6 Implementácia problému ruksaku

Táto kapitola obsahuje detailný popis riešenia problémov popísaných v predošlej kapitole. Podobne ako predošlá kapitola, aj táto je členená na podkapitoly venujúce sa jednotlivým problémom. Každá podkapitola popisuje riešenie jedného problému a korešponduje s príslušnou podkapitolou predošlej kapitoly, ktorá obsahuje popis tohto problému.

U niektorých problémov sa zvolené riešenie líši od pôvodne navrhnutého riešenia. Vo väčšine prípadov je ale použité navrhované riešenie, respektíve jedno z navrhovaných riešení. V takom prípade sa môže popis riešenia odkazovať na popis problému v predošlej kapitole, kde je to-ktoré riešenie podrobnejšie popísané. Niektoré podkapitoly sú doplnené príkladmi zdrojového kódu na ilustráciu konkrétneho riešenia.

Súčasťou výsledného programu je tiež dokumentácia. Tá je automaticky generovaná z komentárov zdrojového kódu pomocou programu `make`. Po zadaní príkazu `make` v koreňovom adresári Python implementácie je v zložke `doc` vytvorená projektová dokumentácia. Táto dokumentácia je vo formáte HTML, je teda možné ju zobraziť pomocou webového prehliadača. Generovaná je programom `pydoc` [31], ktorý musí byť dostupný pri zadaní príkazu `make`. Na spustenie samotného Python programu však nie je program `pydoc` nutný. Príkaz `make` v tomto projekte teda neslúži na kompiláciu zdrojových kódov, ale iba na zjednodušenie generovania dokumentácie. `Makefile` obsahuje tiež cieľ `clean`, ktorý odstráni vygenerovanú dokumentáciu spolu so súbormi obsahujúcimi preložený Python bajtkód. Bližšie informácie sú k dispozícii v komentároch, ako u zdrojových súborov jazyka Python, tak u `Makefile`.

Hlavným súborom programu je súbor `main.py`. Po spustení s argumentom `--help` sa zobrazia informácie o použití programu, ako napríklad podporované prepínače a argumenty a ich predvolené hodnoty. Pri chybe je na štandardný výstup vypísaná chybová hláška a program skončí s nenulovou návratovou hodnotou.

6.1 Obecný postup

Pôvodný C++ kód bol prepísaný do jazyka Python bez väčších logických zmien. Dvojice hlavičkových a implementačných súborov boli nahradené jedným súborom, ktorého obsah korešponduje s pôvodným C++ kódom. Pridaný bol súbor `exc.py`, ktorý obsahuje definíciu pomocnej triedy `Exc`. Tá reprezentuje objekt výnimky a obsahuje pomocnú funkcionalitu. Súbory obsahujúce CUDA C++ kód, používaný z Python kódu, sa nachádzajú v zložke `C++`.

Názvy súborov a identifikátory boli zmenené, aby korešpondovali s konvenciami názvov v jazyku Python [32]. Nové názvy súborov pozostávajú iba z malých písmen oddelených podčiarnikmi. Napríklad zo súborov `GPU_Population.h` a `GPU_Population.cu` vznikol súbor `gpu_population.py`. Boli zmenené tiež názvy tried, aby neobsahovali podčiarniky. Napríklad trieda `TGPU_Population` bola premenovaná na `GPUPopulation`. Názvy metód obsahujú iba malé písmená oddelené podčiarnikmi, napríklad názov `CopyOutIndividual` bol zmenený na `copy_out_individual`.

Jazyk Python neobsahuje ekvivalent kľúčových slov jazyka C++ `private` a `protected`. Existuje iba konvencia, na základe ktorej sú súkromné metódy a atribúty reprezentované tak, že ich názov začína podčiarnikom [33]. Napríklad metóda viditeľnosti `protected` s názvom `RunEvolutionCycle` bola premenovaná na `_run_evolution_cycle`.

Program bol pôvodne implementovaný v jazyku Python verzie 3, ktorá je inštalovaná na školskom výpočtovom serveri SC-GPU1 (viď kap. 2.2). Keďže má byť program testovaný tiež na superpočítači Anselm, je nutné aby ho bolo možné spustiť aj na tomto počítači. Na počítači

Anselm je však knižnica PyCUDA dostupná iba pre jazyk Python verzie 2. Pre verziu 3 dostupná nie je. Preto bol zdrojový kód upravený tak, aby bol kompatibilný s verziou 2 aj 3. Program je tak možné spustiť ako na školskom výpočtovom serveri SC-GPU1, tak na superpočítači Anselm.

6.2 Triedy obsahujúce štruktúry

Podľa pôvodného návrhu mala byť na riešenie tohto problému použitá knižnica struct (viď korešpondujúca podkapitola 5.2). Nakoniec však bolo zvolené iné riešenie.

V Python verzii je značne využívaná knižnica numpy [15]. Predovšetkým je používaná na uchovanie hodnôt cien a váh predmetov. Na to slúži trieda `numpy.ndarray`, ktorá umožňuje efektívne uložiť číselné hodnoty a preniesť ich na GPU pomocou funkcií knižnice PyCUDA. Tieto číselné hodnoty sú uchované vo formáte kompatibilnom s jazykom C/C++, takže je možné ich priamo preniesť na GPU bez potreby konverzie. Zároveň je možné nad týmito hodnotami vykonávať aritmetické operácie, ako napríklad sčítanie, násobenie, apod. Toto by za použitia knižnice struct bolo možné iba po prevedení binárnych hodnôt na typ jazyka Python. Pri použití knižnice numpy je možné s hodnotami narábať ako so štandardnými typmi jazyka Python, pretože tieto hodnoty majú špeciálny typ. Ten je poskytovaný knižnicou numpy a zabezpečuje uloženie hodnôt v podobe kompatibilnej s C/C++. Zároveň tiež poskytuje zmienené operácie nad hodnotami. Dostupné typy sú napríklad `numpy.int32` (korešpondujúci s typom `int` v C/C++) alebo `numpy.float32` (korešpondujúci s typom `float`).

Knižnica numpy poskytuje tzv. štruktúrované polia – *Structured Arrays* (ďalej len SA). Štandardne je možné do poľa typu `numpy.ndarray` uložiť iba primitívne dátové typy, napríklad `numpy.int32`. SA umožňujú vytvoriť pole štruktúrovaného dátového typu. Tento štruktúrovaný dátový typ je heterogénna dátová štruktúra korešpondujúca s typom `struct` v jazyku C++. Poskytuje teda presne takú funkcionalitu, ktorá je v tomto projekte potrebná. Navyše je výhodou, že táto funkcionalita je poskytovaná knižnicou numpy, ktorá už je v projekte použitá. Pri použití knižnice struct by mohli vzniknúť komplikácie, pretože by bolo potrebné zaistiť spoluprácu medzi knižnicami numpy a struct. Tým, že je na všetky účely jednotne použitá knižnica numpy, sa implementácia zjednodušila.

Na použitie SA je najprv potrebné definovať štruktúrovaný dátový typ. Knižnica numpy umožňuje definovať takýto typ pomocou konštruktoru triedy `numpy.dtype`. Táto definícia je podobná definícii typu `struct` v jazyku C++. Je teda možné jednoducho prepísať definíciu z pôvodného C++ kódu do jazyka Python. Kód 6.1 obsahuje definíciu štruktúry `TEvolutionParameters` v jazyku C++.

```
struct TEvolutionParameters {
    int PopulationSize;
    int OffspringPopulationSize;
    int ChromosomeSize;
    int NumOfGenerations;

    float CrossoverPst;
    float MutationPst;
    unsigned int CrossoverUINTBoundary;
    unsigned int MutationUINTBoundary;

    int StatisticsInterval;
    int IntBlockSize;
};
```

Kód 6.1: Definícia štruktúry v jazyku C++.

Korešpondujúci štruktúrovaný dátový typ knižnice numpy je definovaný v kóde 6.2.

```
TEvolutionParameters = numpy.dtype([
    ('PopulationSize',          numpy.int32),
    ('OffspringPopulationSize', numpy.int32),
    ('ChromosomeSize',          numpy.int32),
    ('NumOfGenerations',        numpy.int32),

    ('CrossoverPst',          numpy.float32),
    ('MutationPst',           numpy.float32),
    ('CrossoverUINTBoundary', numpy.uint32),
    ('MutationUINTBoundary',  numpy.uint32),

    ('StatisticsInterval', numpy.int32),
    ('IntBlockSize',       numpy.int32),
])
```

Kód 6.2: Definícia štruktúry v jazyku Python použitím `numpy.dtype`.

Po definovaní dátového typu je možné vytvárať hodnoty tohto typu a prenášať tieto hodnoty medzi CPU a GPU. Prenos dát z CPU na GPU prebieha následovne:

1. Vytvorenie hodnoty typu `list` alebo `tuple`, ktorá obsahuje položky štruktúry. Pre ilustráciu viď kód 6.3.

```
evolution_parameters = (
    population_size,
    offspring_population_size,
    ...
)
```

Kód 6.3: Vytvorenie hodnoty typu `tuple`. Jednotlivé položky tejto hodnoty sú hodnotami prvkov štruktúry. V definícii hodnoty `tuple` sa položky vyskytujú v rovnakom poradí, ako v definícii štruktúry (definícia viď kód 6.2).

2. Prevedenie hodnoty na pole typu `numpy.ndarray` s využitím definovaného štruktúrovaného typu `TEvolutionParameters`. Viď kód 6.4.

```
host_params = numpy.array([evolution_parameters],
                          dtype=TEvolutionParameters)
```

Kód 6.4: Prevedenie hodnoty typu `tuple` na hodnotu typu `numpy.ndarray` s využitím štruktúrovaného typu `TEvolutionParameters`. Týmto je do premennej `host_params` uložené štruktúrované pole. V tomto prípade má pole iba jeden prvok, ktorým je štruktúra typu `TEvolutionParameters`. Tá obsahuje hodnoty špecifikované v `evolution_parameters` (viď kód 6.3). Toto prevedenie na jednoprvkové pole je nutné pre prenos na GPU. Pri prenose je totiž potrebná inštancia `numpy.ndarray`.

3. Prenos vytvoreného poľa na GPU pomocou funkcie `pycuda.driver.memcpy_htod`. Viď kód 6.5.

```
pycuda.driver.memcpy_htod(device_params, host_params)
```

Kód 6.5: Prenos štruktúrovanej hodnoty na GPU. Argument `device_params` je ukazovateľom do pamäti GPU a ukazuje na štruktúru rovnakého typu ako `TEvolutionParameters`. Argument `host_params` je typu `numpy.ndarray` (viď kód 6.4).

Pred prenosom je možné vytvorené pole registrovať ako „pinned“ pamäť použitím funkcie `pycuda.driver.register_host_memory`. Pri prenose opačným smerom sa postupuje obdobne.

K položkám štruktúry sa pristupuje rovnako ako k položkám vstavanej triedy `dict`. Predtým je potrebné túto štruktúru získať z poľa, čo znázorňuje kód 6.6.

```
params_struct = host_params[0]
population_size = params_struct['PopulationSize']
```

Kód 6.6: Získanie štruktúrovanej hodnoty `params_struct` z poľa `host_params` a následné získanie položky štruktúry. `host_params` je pole typu `numpy.ndarray`, `params_struct` je štruktúrovaná hodnota typu `TEvolutionParameters` (viď kód 6.2).

Na GPU sa pristupuje k položkám štruktúry bežným spôsobom. Tým pádom netreba meniť spôsob použitia týchto štruktúr v kerneloch.

6.3 Definície štruktúr v hlavičkových súboroch

Pri tomto probléme bolo použité jedno z riešení navrhovaných v podkapitole 5.3. Definície štruktúr, dátových typov a konštánt boli presunuté z pôvodných súborov do hlavičkového súboru `CUDA_Kernels.h`. V tomto súbore je pri definíciách v komentároch vyznačené, z ktorého súboru tá-ktorá definícia pochádza. Výsledné C++ súbory použité v projekte sú tak iba `CUDA_Kernels.h` a `CUDA_Kernels.cu`.

6.4 Globálny zámok na GPU

Pôvodný návrh bol rozdeliť triedu `TGPU_Lock` na CPU časť a GPU časť. Táto trieda je v pôvodnom kóde používaná v metóde `TGPU_Statistics::Calculate`. Pri volaní tejto metódy je vytvorený zámok `TGPU_Lock`. Následne je tento zámok predaný kernelu `CalculateStatistics` a po jeho ukončení je zámok dealokovaný. Táto opakovaná alokácia a dealokácia nie je potrebná. Zámok možno alokovať raz, pri vytvorení inštancie triedy `TGPU_Statistics`, a uložiť ho ako inštančnú premennú v tejto triede. Pri volaní metódy `Calculate` sa použije vždy ten istý zámok a v deštruktore triedy bude zámok dealokovaný.

Trieda `TGPU_Lock` bola v Python verzii odstránená. Namiesto nej je v konštruktore triedy `GPUStatistics` alokovaná hodnota typu `int` na GPU. Ukazovateľ na túto hodnotu je uložený ako inštančná premenná. V deštruktore je potom táto hodnota dealokovaná. V metóde `calculate` je ukazovateľ predaný kernelu `CalculateStatistics`, ktorý ho potom ďalej predá funkciám `Lock` a `Unlock`. Tie boli prepísané z metód triedy `TGPU_Lock` na GPU funkcie, tak ako je popísané v návrhu (podkapitola 5.4).

6.5 Singleton triedy

Singleton trieda `TParameters` bola prepísaná do Python verzie s čo najmenšími funkčnými zmenami. Výsledná trieda `Params` obsahuje triednu premennú `__instance`, ktorá je počiatočne prázdna (jej hodnota je nastavená na `None`). Pri prvom zavolaní triednej metódy `instance` je vytvorená inštancia, ktorá je uložená do premennej `__instance`. Pri ďalšom volaní metódy je vrátená už existujúca inštancia. Metódy a atribúty tejto singleton inštancie sú inštančné, tak ako

u bežnej triedy. Teda všetky atribúty a metódy triedy `Params` sú inštančné, okrem atribútu `__instance` a metódy `instance`, ktoré sú triedne.

Navrhovaná alternatívna možnosť implementácie vzoru singleton (viď podkapitola 5.5) bola aplikovaná pri riešení problému uchovania preložených kernelov. Ten je popísaný v nasledujúcej podkapitole.

6.6 Preklad a spúšťanie kernelov

Tento problém bol vyriešený použitím vzoru singleton, ako bolo navrhnuté v podkapitole 5.6. Na správu kernelov slúži trieda `Kernels` definovaná v súbore `kernels.py`. Táto trieda je však implementovaná odlišne od triedy `Params`, popísanej v predošlej podkapitole.

Implementácia triedy `Params` vyžaduje vytvorenie a uchovanie singleton inštancie. Kód, ktorý túto inštanciu používa, ju musí najprv získať volaním špeciálnej triednej metódy `instance`. Nie je však potrebné, aby existovala inštancia – stačí, ak budú všetky metódy a atribúty triedne.

Tento prístup bol použitý pri implementácii triedy `Kernels`. Tá neobsahuje triedny atribút s inštanciou, ani triednu metódu na jej získanie. Všetky metódy a atribúty, ktoré by inak boli inštančné, sú triedne. Odpadá tým nutnosť vytvárať a udržiavať inštanciu a klientský kód nemusí volať metódu na získanie tejto inštancie. Namiesto toho pracuje priamo s triedou, tak ako by to bola singleton inštancia.

Táto trieda obsahuje triednu metódu `compile`, ktorá je volaná pri spustení programu. Jej volaním sa preložia kernely definované v súbore `CUDA_Kernels.cu`. Výsledkom prekladu je PyCUDA modul, ktorý sa uloží do triednej premennej. Z neho sú následne pomocou metódy `get_function` získané kernely a tie sú tiež uložené ako triedne premenné. Na získanie globálneho symbolu, ako napríklad konštanty, slúži metóda `get_global`. Táto metóda iba zavolá rovnomennú metódu uloženého modulu a vráti výsledok. Ako argument dostane názov symbolu a vráti ukazovateľ na daný symbol. Ten je potom možné použiť napríklad pre kopírovanie dát na GPU pomocou funkcie `pycuda.driver.memcpy_htod`. Kód 6.7 znázorňuje použitie triedy `Kernels`.

```
from kernels import Kernels

Kernels.compile()

Kernels.calculate_statistics(
    statistics_data, population_data, lock,
    grid=GRID_DIMENSIONS,
    block=BLOCK_DIMENSIONS
)
```

Kód 6.7: Použitie triedy `Kernels`: preklad zdrojového kódu a spustenie kernelu `CalculateStatistics`. Názvy zdrojových súborov sú definované v triede `Kernels`.

6.7 Meranie trvania výpočtu

Na meranie výkonu kernelov sú použité CUDA events, ako bolo navrhnuté v podkapitole 5.7. Toto meranie bolo implementované jednak v Python kóde, a tiež existujúci C++ kód bol upravený tak, aby namiesto funkcie `clock` používal CUDA events.

Meranie režijných nákladov obslužného kódu je realizované odlišne od pôvodného návrhu. Podľa návrhu mal byť celkový čas behu programu meraný priamo programom samotným. Tento prístup ale nie je objektívny.

Pri použití pôvodne navrhnutého postupu by bolo meranie času zahájené až vo funkcii `main`. V Python verzii je ale vykonaných niekoľko operácií ešte predtým, než je funkcia `main` zavolaná. Menovite sú vykonané všetky príkazy `import`, z ktorých niektoré sa významne podieľajú na celkovom trvaní behu programu. To je badateľné hlavne pri malej veľkosti populácie a nízkom počte generácií. Obzvlášť dlho trvá vykonanie príkazu `import pycuda.autoinit`. V testovacom prostredí (počítače SC-GPU1, Anselm) trvá vykonanie tohto príkazu stovky milisekúnd až jednotky sekúnd. Je to zrejme preto, lebo v ňom dochádza k inicializácii GPU, čo je evidentne časovo náročná operácia.

Pôvodný prístup tiež nezahrňuje čas potrebný na inicializáciu programu pred jeho spustením. Tým sa myslí načítanie programu do pamäte a vykonanie inicializácie na úrovni systému. V C++ verzii je táto inicializačná fáza závislá na použitom prekladači, štandardnej knižnici, operačnom systéme a ďalších faktoroch [34]. Vo verzii Python túto fázu tvorí inicializácia interpretu, ktorá je podobná ako u C++ verzie. Patrí sem tiež spracovanie zdrojových súborov, teda ich preklad do bajtkódu. Táto inicializačná fáza ale nemá veľký vplyv na celkové trvanie, vzhľadom na relatívne malý rozsah programu.

Z týchto dôvodov bolo od merania času priamo v programe upustené. Namiesto toho bude čas meraný externým nástrojom, ktorý zmeria celkový čas od spustenia programu po jeho ukončenie. Tento prístup umožňuje objektívnejšie porovnať režijné náklady verzii C++ a Python. Tiež bližšie vystihuje vnímanie trvania behu programu z pohľadu užívateľa. Na meranie bude použitý nástroj `time` [35] dostupný v POSIX-kompatibilných operačných systémoch.

6.8 Rozsah kódu

Verzia C++ pozostáva z trinástich zdrojových súborov. Z nich dva obsahujú kód kernelov a zvyšných jedenásť súborov obsahuje obslužný kód. Celkový počet riadkov kódu v týchto jedenástich súboroch je 2149. Ich celková veľkosť je 61570 bajtov.

Verzia Python obsahuje deväť súborov s obslužným kódom. Počet riadkov kódu je 1229 a veľkosť súborov je 32856 bajtov. V porovnaní s C++ verziou je počet riadkov kódu nižší o 43 %. Veľkosť súborov je nižšia o 53 %.

Je niekoľko dôvodov, prečo je tento rozdiel tak veľký. Jednak jazyk Python obsahuje množstvo vstavaných funkcií uľahčujúcich často vykonávané úkony. Zároveň je prirodzene expresívnejší a stručnejší v syntaxi než C++. Vhodným príkladom na ilustráciu je napríklad cyklus iterujúci cez číselné hodnoty. Ten je v jazyku C++ možné zapísať niekoľkými spôsobmi. Kód 6.8 znázorňuje najjednoduchší spôsob bez použitia externej knižnice.

```
for (int i = 0; i < N; i++)
{
    statement();
}
```

Kód 6.8: `for` cyklus v jazyku C++.

V jazyku Python je tá istá konštrukcia omnoho jednoduchšia, s využitím vstavanej funkcie `range`. Viď kód 6.9.

```
for i in range(N):
    statement()
```

Kód 6.9: `for` cyklus v jazyku Python.

Ďalším dôvodom je duplicita zdrojového kódu u C++. Takmer každý implementačný súbor totiž má svoj hlavičkový súbor. V hlavičkovom súbore sú napríklad definície tried, typov a konštánt. V definíciách tried, prípadne tiež štruktúr, sú deklarácie metód. Tieto metódy sú potom implementované v implementačných súboroch. To vedie k duplikácii kódu, keďže prototyp metódy sa nachádza aj v hlavičkovom, aj v implementačnom súbore. Duplikované sú taktiež komentáre, keďže väčšinou je komentovaná aj deklarácia, aj definícia metódy. Zvyšuje sa tak celkový objem zdrojového textu, ako aj počet zdrojových súborov.

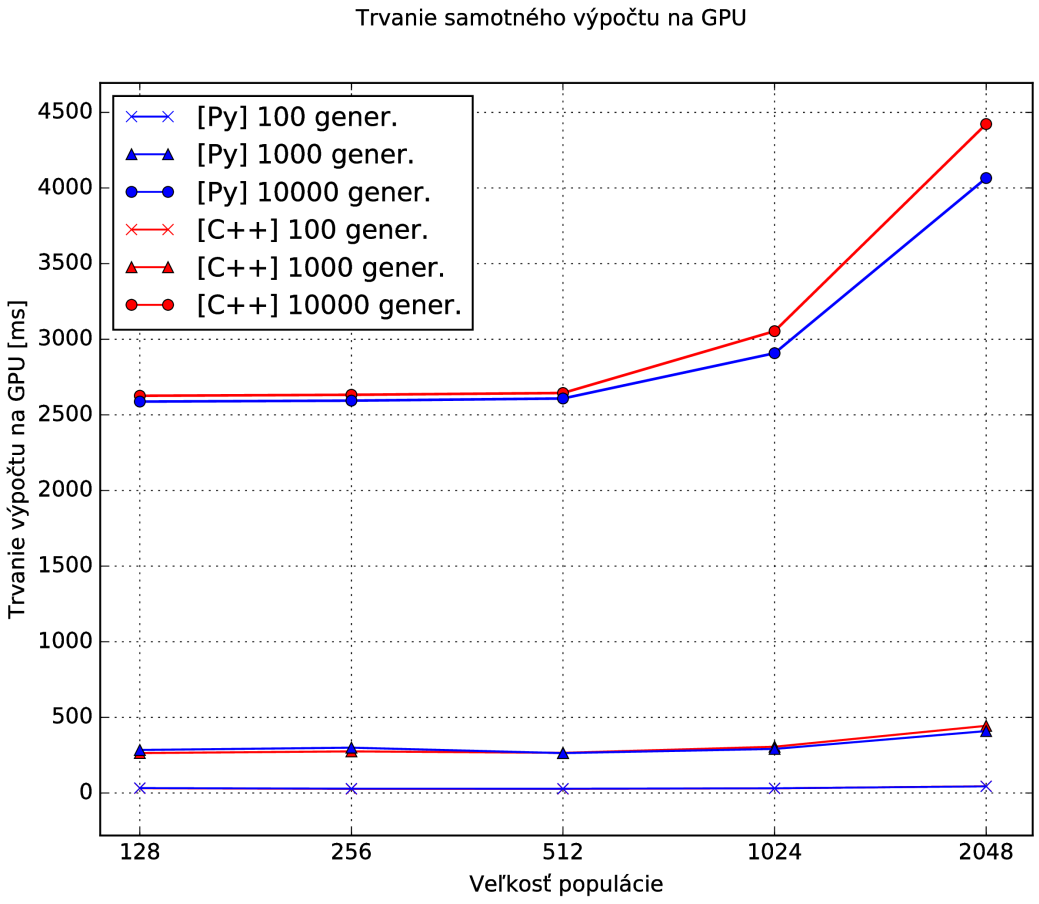
Keďže C++ je prekladaný jazyk, je pri jeho použití potrebné zabezpečiť preklad kódu. Nutnosť prekladu je ďalším prvkom, ktorý zväčšuje rozsah kódu. Na preklad sa často používa program `make` [36] alebo podobný. Závislosti medzi jednotlivými súbormi a spôsoby ich prekladu sú definované v súbore `Makefile`, ktorých sa v projekte môže vyskytovať niekoľko. Tieto súbory tvoria iba malý podiel zdrojového kódu, no i tak zvyšujú jeho rozsah a pre programátora predstavujú prácu navyše. Jazyk Python nevyžaduje na spustenie programu preklad kódu, teda súbory ako `Makefile` nie sú potrebné.

7 Testovanie

Za účelom overenia funkčnosti a porovnania výkonu verzií C++ a Python bolo vykonané testovanie. To prebiehalo predovšetkým na školskom serveri SC-GPU1, ktorý je vybavený modernejšími grafickými kartami než superpočítač Anselm (špecifikácie sú uvedené v kapitole 2.2). Taktiež práca na tomto serveri je jednoduchšia v porovnaní s počítačom Anselm, keďže je možné pracovať s grafickými kartami priamo zo štandardnej SSH konzoly. Na počítači Anselm je pre prácu s grafickými kartami nutné použiť Portable Batch System, čo prácu s týmto počítačom komplikuje (viď kapitola 2.2).

Pre účely testovania bol existujúci C++ kód rozšírený o možnosť použitia pevného náhodného semienka. Je to preto, aby bolo možné porovnať výsledky C++ verzie s výsledkami Python verzie. Taktiež bol C++ kód upravený tak, aby používal CUDA events na meranie trvania výpočtu na GPU. Rovnaký mechanizmus je použitý v Python verzii. Na meranie celkového trvania behu programu bol použitý nástroj `time` [35].

Cieľom merania je objektívne porovnať časovú náročnosť verzií C++ a Python z pohľadu užívateľa. Zároveň musia byť výsledky merania dostatočne jednoduché a zrozumiteľné. Preto bude okrem trvania behu kernelov na GPU porovnané iba celkové trvanie programu, od spustenia po ukončenie. Tento čas sa nazýva reálny čas alebo tiež *„wall clock time“*.

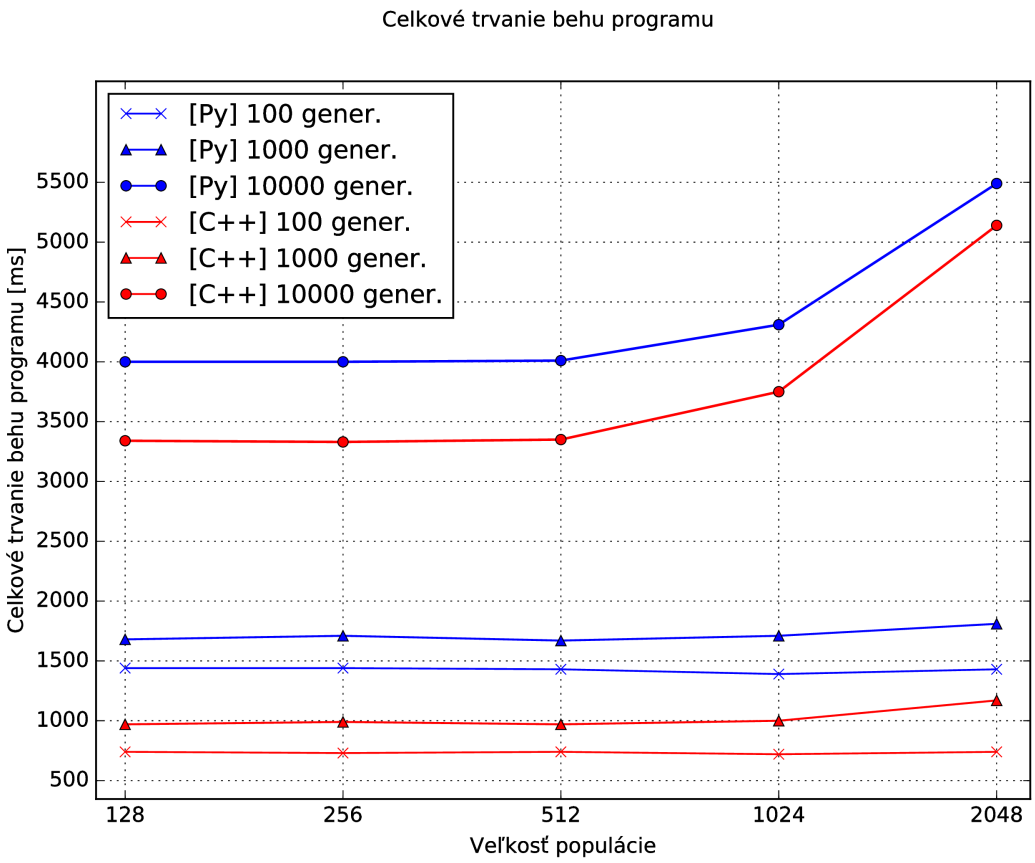


Graf 7.1: Čistý čas výpočtu na GPU u verzií C++ a Python v závislosti na počte generácií a veľkosti populácie. Verzia Python je pri väčších vstupoch mierne výkonnejšia.

Pri meraní bol použitý vstup obsahujúci 10-tisíc predmetov (súbor `knap_10000.txt`). Ostatné vstupy, obsahujúce 10, respektíve 40 predmetov, boli príliš malé na to, aby na nich bolo možné efektívne porovnať výkon. Štatistiky boli obmedzené iba na konečný výpis po skončení výpočtu. Pri všetkých meraniach bolo použité rovnaké náhodné semienko. Vďaka tomu bolo možné potvrdiť, že Python verzia funguje správne a dosahuje výsledky rovnakej kvality ako verzia C++.

Graf 7.1 ukazuje trvanie behu kernelov pre rôzne počty generácií a veľkosti populácie. Kód kernelov je u verzií C++ a Python takmer rovnaký. Preto by sa trvanie behu kernelov u jednotlivých verzií nemalo príliš líšiť. Pri 100 generáciách je výkon Python verzie priemerne o 3,93 % nižší než výkon verzie C++. Pri 1000 generáciách je rozdiel voči C++ verzii iba 0,7 %. Pri veľkom počte generácií je výkon Python verzie dokonca o niečo vyšší, než výkon verzie C++. Konkrétne pri 10000 generáciách je Python verzia o 3,43 % výkonnejšia, než C++. To môže byť spôsobené niekoľkými rozdielmi medzi C++ a Python verziou.

Jednak sú určité malé rozdiely v samotných kerneloch, napríklad zamykanie a odomykanie globálneho zámku (viď kapitola 6.4). Na to slúžia operácie `Lock` a `Unlock`, ktoré sú v C++ verzii implementované ako metódy, kým v Python verzii sú to funkcie. To by ale nemalo mať veľký vplyv na výkon, keďže štatistiky sa počas testovacieho výpočtu počítajú iba raz, na konci evolúcie. Ďalším rozdielom je, že zámok je v Python verzii alokovaný iba raz za celý beh programu. Vo verzii C++ je alokovaný pri každom výpočte štatistík. Tento rozdiel by takisto nemal mať veľký vplyv. Rozdielny je tiež spôsob alokácie parametrov programu v pamäti CPU. Pre štruktúru obsahujúcu parametre je vo verzii C++ použitá štandardná pamäť, ale vo verzii Python je použitá „pinned“ pamäť. Keďže sa parametre prenášajú z CPU do GPU iba raz pred spustením výpočtu, ani tento rozdiel by nemal mať značný vplyv na výkon.

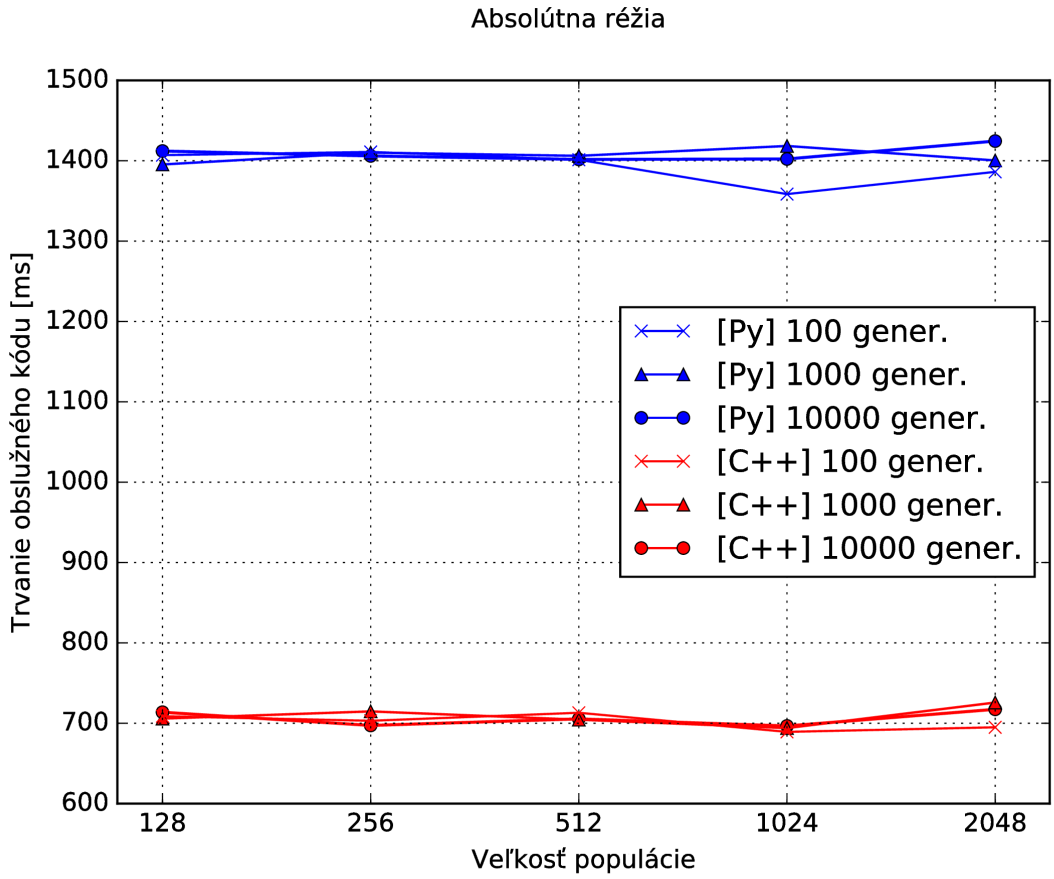


Graf 7.2: Celkové trvanie programu na GPU u verzií C++ a Python v závislosti na počte generácií a veľkosti populácie.

Okrem rozdielov v zdrojovom kóde môžu byť rozdiely tiež v spôsobe prekladu a vo výslednom strojovom kóde. V Python verzii sú použité takmer rovnaké prepínače prekladača `nvcc` ako vo verzii C++. Jediným odlišným prepínačom je `–device-c` [30], ktorý je použitý vo verzii C++, ale v Python verzii ho nie je možné použiť. Tento prepínač má význam iba pri generovaní spustiteľných súborov, na výkon pravdepodobne nemá vplyv. Knižnica PyCUDA [13] však môže volať prekladač iným spôsobom, napríklad pridaním ďalších prepínačov, alebo môže iným spôsobom spúšťať kernely.

Graf 7.2 ukazuje celkové trvanie behu programu pre rôzne počty generácií a veľkosti populácie. Celkové trvanie je u verzie Python v zásade dlhšie, než u verzie C++. To je dané vyššími režijnými nákladmi Python verzie. Kým pri 100 generáciách je rozdiel takmer dvojnásobný, pri 1000 generáciách je Python verzia pomalšia približne o dve tretiny. Pri 10000 generáciách je priemerný rozdiel už iba zhruba jedna šestina trvania C++ verzie. To, že sa relatívny rozdiel medzi C++ a Python verziou zmenšuje, je dané tým, že režijné náklady sú konštantné, kým trvanie samotného výpočtu je závislé na veľkosti vstupu. Tú určuje napr. počet predmetov, veľkosť populácie, počet generácií atp. Ďalším dôvodom je vyšší výkon GPU kódu u Python verzie.

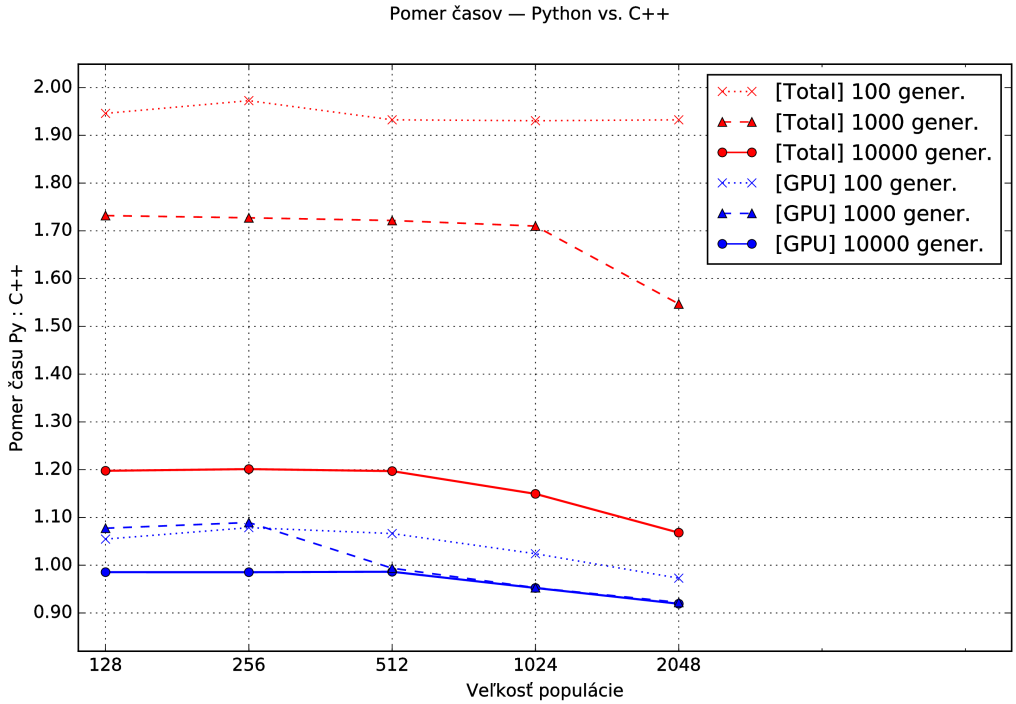
Graf 7.3 ukazuje, že režijné náklady sú u oboch verzií približne konštantné. U verzie C++ trvá vykonanie obslužného kódu v priemere 706 milisekúnd, u verzie Python 1403 milisekúnd. Režijné náklady Python verzie sú priemerne o 98,7 % vyššie.



Graf 7.3: Režijné náklady u verzií C++ a Python v závislosti na počte generácií a veľkosti populácie. Režijné náklady tvorí vykonanie obslužného kódu. Spolu s trvaním behu kernelov na GPU tvoria celkové trvanie behu programu.

Graf 7.4 znázorňuje pomer výkonu verzií Python a C++. Režijné náklady sú u oboch verzií konštantné, preto sa ich podiel na celkovom trvaní s rastúcou veľkosťou vstupu zmenšuje. U veľkých vstupov (veľká populácia, mnoho generácií) tvorí výpočet na GPU väčšinu celkového trvania. Zároveň je samotný výpočet pri veľkých vstupoch približne rovnako rýchly u oboch verzií. To znamená, že so zväčšujúcimi sa vstupmi sa zmenšuje výkonový rozdiel medzi verziami C++ a Python. Kým pri evolúcii o 100 generáciách je celkové trvanie Python verzie priemerne o 94,3 % dlhšie, pri 1000 generáciách je to už iba 68,7 %. Pri 10000 generáciách je Python verzia pomalšia priemerne o 16,3 %, pričom so zväčšujúcou populáciou sa rozdiel zmenšuje. Pri populácii o 2048 jedincoch je Python verzia pomalšia iba o 6,8 %.

To, že sa rozdiel zmenšuje, je dané hlavne tým, že režijné náklady tvoria čoraz menšiu časť celkového trvania programu. V prospech Python verzie je tiež fakt, že pri väčších vstupoch je u nej výpočet rýchlejší než u C++ verzie. Pri 100 generáciách je síce u Python verzie výpočet priemerne o 3,92 % pomalší, no pri 10000 generáciách je o 3,43 % rýchlejší.



Graf 7.4: Pomer trvania verzie Python k verzii C++ v závislosti na počte generácií a veľkosti populácie. S rastúcou veľkosťou vstupu sa pomer celkového trvania verzie Python k verzii C++ blíži k 1.

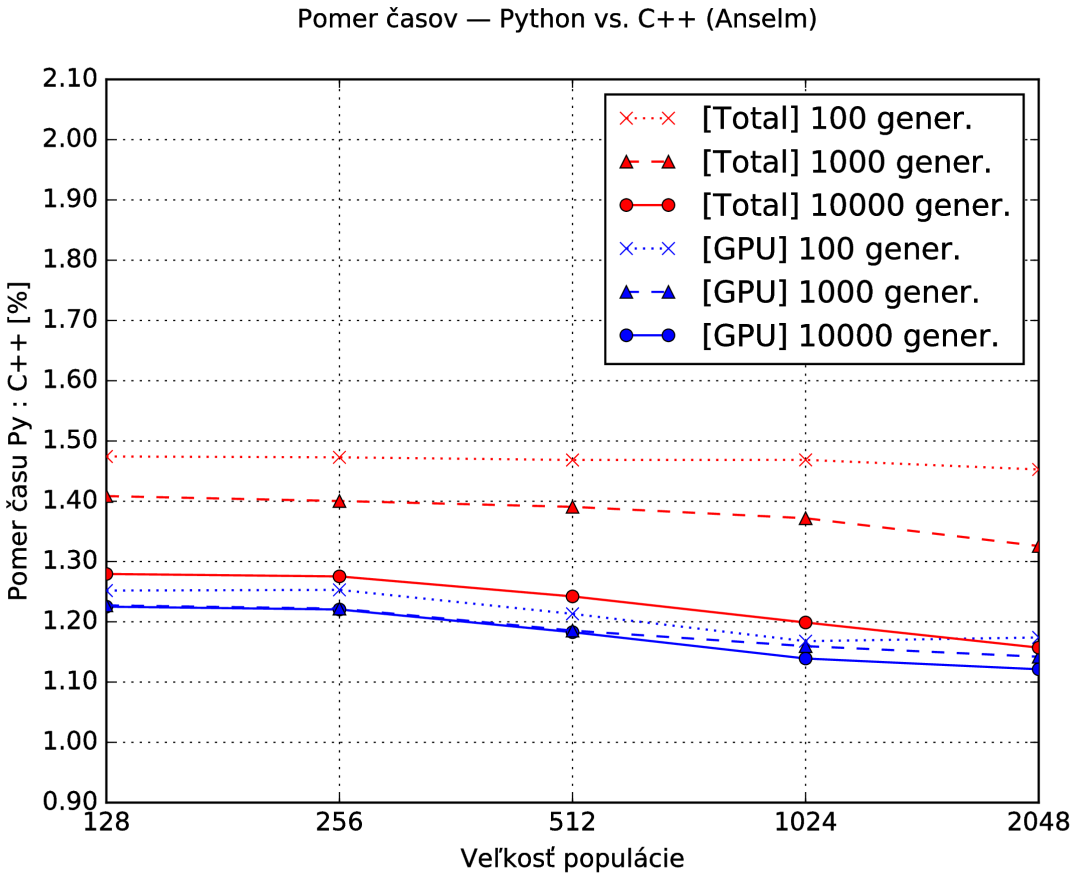
Testovanie bolo v obmedzenom rozsahu vykonané tiež na superpočítači Anselm. Vzhľadom na to, že jeho hardvérové vybavenie je v porovnaní so školským počítačom SC-GPU1 relatívne zastaralé, nebol na tieto testy kladený veľký dôraz. Pri testovaní bol použitý modul `ScientificPython/2.9.4-intel-2016.01-Python-2.7.9`.

Výsledky získané na počítači Anselm sú značne odlišné od výsledkov získaných na počítači SC-GPU1. Celkový výkon oboch verzií je omnoho nižší. To je dané hlavne tým, že počítač Anselm je vybavený staršími grafickými kartami NVIDIA Kepler K20, kým počítač SC-GPU1 je vybavený novšími a výkonnejšími kartami NVIDIA GeForce GTX 1080 (popísané v kap. 2.2). Veľké rozdiely sú ako v trvaní samotného výpočtu, tak v režijných nákladoch. Celková charakteristika meraných veličín ale zväčša súhlasí s meraním vykonaným na počítači SC-GPU1. Závislosti veličín sú takmer rovnaké. Napríklad režijné náklady sú konštantné, trvanie výpočtu rastie približne lineárne počtom generácií, atď. Rozdiely medzi výkonom C++ a Python verzie sú však odlišné.

Trvanie obslužného kódu je u verzie Python priemerne 2368 ms, v porovnaní s SC-GPU1 o 965 ms viac (+68,8 %). U verzie C++ je to 1598 ms, v porovnaní s SC-GPU1 o 892 ms viac (+126,6 %). Tým sa znížil relatívny rozdiel v režijných nákladoch u jednotlivých verzií. Na počítači Anselm je réžia u verzie Python vyššia iba o 48,2 %, kým na počítači SC-GPU1 je rozdiel takmer dvojnásobný.

U veľkých vstupov však réžia nemá veľký vplyv. Dôležitejší je výkon kernelov, respektíve trvanie samotného výpočtu na GPU. To by teoreticky malo byť približne rovnaké u oboch verzií, no ako vidno na grafe 7.5, nie je tomu tak. Na počítači Anselm je výkon kernelov u Python verzie vždy nižší, na rozdiel od SC-GPU1, kde bol v niektorých prípadoch vyšší. To môže byť spôsobené odlišnou verziou jazyka Python či knižnice PyCUDA. Absolútny rozdiel vo výkone kernelov sa zväčšuje s rastúcim vstupom. Relatívny rozdiel sa ale zmenšuje, ako je vidno na grafe 7.5. Pri 100 generáciách je tento rozdiel priemerne 21,2 %, pri 1000 generáciách 18,7 % a pri 10000 generáciách 17,8 %.

Celkové trvanie programu je na počítači Anselm značne dlhšie. Pomer celkového trvania Python verzie ku C++ verzii sa však zmenšuje, rovnako ako na SC-GPU1. Tento pomer je pri nízkom počte generácií v porovnaní s SC-GPU1 menší. To je dané vyššou réžiou v porovnaní s SC-GPU1, ktorá pri nízkom počte generácií tvorí podstatnú časť celkového času. Pri 100 generáciách je celkové trvanie Python verzie v priemere o 46,7 % vyššie. Na SC-GPU1 bol rozdiel 94,3%. Pri 1000 generáciách je tento rozdiel 37,9 %; na SC-GPU1 to bolo 68,7 %. Pri 10000 generáciách je ale tento rozdiel väčší v porovnaní s SC-GPU1. Python verzia je pri 10000 generáciách pomalšia o 23,1 %, kým na SC-GPU1 bola pomalšia iba o 16,3 %. Konkrétne pri populácii o 2048 jedincoch je potom pomalšia o 15,7 %, resp. 6,8 % na SC-GPU1.



Graf 7.5: Pomer trvania verzie Python k verzii C++ v závislosti na počte generácií a veľkosti populácie. S rastúcou veľkosťou vstupu sa pomer celkového trvania verzie Python k verzii C++ blíži k 1. Meranie vykonané na počítači Anselm.

8 Záver

V rámci tejto práce bola implementovaná sada mikrotestov a jeden komplexnejší problém. Mikrotesty implementujú menšie problémy v jazykoch C++ a Python, napríklad rýchlosť prenosu dát medzi CPU a GPU. Komplexnejší problém predstavuje reálnu aplikáciu, ktorá je prakticky použiteľná.

Mikrotesty slúžili na oboznámenie sa s programovaním aplikácií s využitím GPU, predovšetkým s platformou CUDA a knižnicou PyCUDA pre jazyk Python. Z pozorovania výsledkov týchto testov bolo vyvodených niekoľko záverov. Pamäť je dobré alokovať ako „pinned“, čo zvyšuje rýchlosť prenosu dát medzi operačnou pamäťou procesoru a pamäťou grafickej karty. Tá je často podstatným časovým nákladom. Rýchlosť pamäte GPU je tiež často obmedzujúcim faktorom a preto treba vhodne využívať pamäť, napr. použitie zdieľanej pamäte, konštantnej pamäte, atp. Čas potrebný na prenos dát je u výpočtovo nenáročných úloh nezanedbateľný, napr. u SAXPY, násobení malých matíc apod. Na GPU sa preto oplatí akcelerovať hlavne výpočtovo intenzívne úlohy.

Komplexnejší problém predstavuje problém ruksaku. Ten bol implementovaný v CUDA C++ pomocou genetického algoritmu akcelerovaného na GPU. V jazykoch C++ a Python bol potom implementovaný obslužný kód, z ktorého sú spúšťané CUDA kernely vykonávajúce samotný výpočet. Zdrojový kód kernelov je v oboch verziách približne rovnaký.

Vývoj prebiehal s použitím systému Git, ktorý je inštalovaný na školskom serveri SC-NAS[2]. Problémy riešené počas implementácie boli evidované v systéme ako *issues*. Návrhy riešení boli uvedené v popise *issue,* v komentári bolo potom popísané konkrétne použité riešenie.

Ako sa ukázalo pri implementácii a testovaní mikrotestov a problému ruksaku, jazyk Python je vhodný na tvorbu aplikácií akcelerovaných na grafickej karte. Výsledky dosiahnuté použitím jazyka Python v kombinácii s knižnicou PyCUDA sú porovnateľné s výsledkami dosiahnutými použitím jazyka C++ v kombinácii s natívnou knižnicou CUDA. Rozdiel vo výkone sa zmenšuje s rastúcou veľkosťou vstupných dát. Pri veľkých, výpočtovo náročných úlohách je teda výkonový rozdiel malý. Práve takéto úlohy sú často akcelerované na grafickej karte.

Úsilie vynaložené na vytvorenie aplikácie v jazyku Python je pritom značne nižšie než v jazykoch C alebo C++. Dokazuje to aj rozsah zdrojového kódu, ktorý je v jazyku Python výrazne menší než v C++ – v niektorých prípadoch takmer polovičný. To prispieva k udržiavateľnosti zdrojového kódu, avšak mínusom je chýbajúca typová kontrola.

Celkovo je jazyk Python vhodnou alternatívou k C/C++, obzvlášť pri malých a stredne veľkých projektoch. Jeho výhody spočívajú v expresívnej a intuitívnej syntaxi, automatickej správe pamäti a množstve dostupných knižníc. Vďaka tomu jeho použitie znižuje dobu vývoja aplikácie a tiež rozsah zdrojového kódu. Tým sa zvyšuje čitateľnosť zdrojového kódu a zlepšuje sa udržiavateľnosť projektu. Hlavnou nevýhodou je chýbajúca typová kontrola, ktorá ho činí nevhodným pre veľké, rozsiahle aplikácie.

2 Adresa servera SC-NAS: `sc-nas.fit.vutbr.cz`

Referencie

[1] *Open MPI*. [online]. [cit. 2017-02-13]. Dostupné z: <https://www.open-mpi.org/>

[2] *OpenCL*. [online]. [cit. 2017-02-13]. Dostupné z: <https://www.khronos.org/opencl/>

[3] BARLAS, Gerassimos. *Multicore and gpu programming: an integrated approach*. ISBN 978-0124171374.

[4] CUDA Parallel Computing Platform. *Nvidia.com*. [online]. [cit. 2017-02-13]. Dostupné z: <http://www.nvidia.com/object/cuda_home_new.html>

[5] *Nvidia.com*. [online]. [cit. 2017-05-15]. Dostupné z: <http://www.nvidia.com/content/global/global.php>

[6] Hardware model (Tesla) of CUDA-enabled GPUs. In: *Springer.com* [online]. [cit. 2017-04-23]. Dostupné z: <https://static-content.springer.com/esm/art%3A10.1186%2F1756-0500-2-73/MediaObjects/13104_2009_211_MOESM3_ESM.png>

[7] HILL, Mark D., Norman P. JOUPPI a Gurindar. SOHI. *Readings in computer architecture*. San Francisco: Morgan Kaufmann, 2000. ISBN 1-55860-539-8.

[8] Anselm Cluster Introduction. *IT4Innovations Documentation*. [online]. [cit. 2016-12-18]. Dostupné z: <https://docs.it4i.cz/anselm/introduction/>

[9] NVIDIA® TESLA® GPU ACCELERATORS. *Nvidia.com*. [online]. [cit. 2016-12-18]. Dostupné z: <http://www.nvidia.com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf>

[10] Secure Shell. In: *Archlinux.org*. [online]. [cit. 2017-05-15]. Dostupné z: <https://wiki.archlinux.org/index.php/Secure_Shell>

[11] NVIDIA® GeForce® GTX 1080 8GB. *Pny.com*. [online]. [cit. 2016-12-18]. Dostupné z: <https://www.pny.com/File%20Library/Support/PNY%20Products/Resource%20Center/Graphics%20Cards/GTX%201000%20Series/PNY-NVIDIA-GeForce-GTX-1080-8GB.pdf>

[12] *Python.org*. [online]. [cit. 2016-12-19]. Dostupné z: <https://www.python.org/>

[13] PyCUDA. *Andreas Klöckner's web page*. [online]. [cit. 2017-01-11]. Dostupné z: <https://mathema.tician.de/software/pycuda/>

[14] Numba. *Pydata.org*. [online]. [cit. 2017-02-13]. Dostupné z: <http://numba.pydata.org/>

[15] *NumPy*. [online]. [cit. 2017-01-11]. Dostupné z: <http://www.numpy.org/>

[16] CUDA Runtime API – CUDA Event Management. *Nvidia.com*. [online]. [cit. 2017-04-26]. Dostupné z: <http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html>

[17] MITCHELL, Melanie. *An introduction to genetic algorithms*. Cambridge, Mass: MIT Press, 1996. ISBN 9780585030944.

[18] KELLERER, Hans, Ulrich PFERSCHY a David PISINGER. *Knapsack problems*. Berlin [u.a.]: Springer, 2010. ISBN 9783642073113.

[19] VAN LEEUWEN, Jan. *Handbook of theoretical computer science: algorithms and complexity*. Massachusetts: MIT Press, 1998. ISBN 0-262-72014-0.

[20] Example of a one-dimensional (constraint) knapsack problem. In: *Wikipedia.org* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-05-10]. Dostupné z: <https://upload.wikimedia.org/wikipedia/commons/f/fd/Knapsack.svg>

[21] CHVÁTAL, Vašek. *Linear programming*. New York: W.H. Freeman, c1983. ISBN 978-0-7167-1587-0.

[22] CLAUSEN, Jens. *Branch and Bound Algorithms—Principles and Examples.* 1999. Technická správa. University of Copenhagen.

[23] CUDA Samples. *Nvidia.com*. [online]. [cit. 2017-01-12]. Dostupné z: <http://docs.nvidia.com/cuda/cuda-samples/>

[24] Schematic depiction of the matrix product AB of two matrices A and B. In: *Wikipedia.org* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-05-10]. Dostupné z: <https://upload.wikimedia.org/wikipedia/commons/e/eb/Matrix_multiplication_diagram_2.svg>

[25] NVIDIA GeForce GTX 1080 and GTX 1070 Officially Announced. *Geeks3D* [online]. [cit. 2017-01-13]. Dostupné z: <http://www.geeks3d.com/20160507/nvidia-geforce-gtx-1080-and-gtx-1070-officially-announced/>

[26] *Git*. [online]. [cit. 2017-05-10]. Dostupné z: <https://git-scm.com/>

[27] __main__ — Top-level script environment. *Python.org*. [online]. [cit. 2017-04-26]. Dostupné z: <https://docs.python.org/3/library/__main__.html>

[28] struct — Interpret bytes as packed binary data. *Python.org*. [online]. [cit. 2017-05-10]. Dostupné z: <https://docs.python.org/3.6/library/struct.html>

[29] GAMMA, Erich. *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, c1995. ISBN 0-201-63361-2.

[30] NVIDIA CUDA Compiler Driver NVCC. *Nvidia.com*. [online]. [cit. 2017-05-15]. Dostupné z: <http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/>

[31] pydoc — Documentation generator and online help system. *Python.org*. [online]. [cit. 2017-05-01]. Dostupné z: <https://docs.python.org/3.6/library/pydoc.html>

[32] PEP 8 -- Style Guide for Python Code. *Python.org*. [online]. [cit. 2017-04-26]. Dostupné z: <https://www.python.org/dev/peps/pep-0008/>

[33] Classes – Python 3.6.1 documentation. *Python.org*. [online]. [cit. 2017-04-27]. Dostupné z: <https://docs.python.org/3.6/tutorial/classes.html>

[34] How main() is executed on Linux LG #84. *Tldp.org*. [online]. [cit. 2017-04-27]. Dostupné z: <http://www.tldp.org/LDP/LG/issue84/hawk.html>

[35] The Open Group Base Specifications Issue 7 – time. *Opengroup.org*. [online]. [cit. 2017-04-28]. Dostupné z: <http://pubs.opengroup.org/onlinepubs/9699919799/utilities/time.html>

[36] GNU Make. *Gnu.org*. [online]. [cit. 2017-05-15]. Dostupné z: <https://www.gnu.org/software/make/>

Príloha A

Obsah CD

- `microtests/` – implementácie mikrotestov
  - `src/` – zdrojové súbory testov
  - `inc/` – pomocné súbory

- `knapsack/` – problém ruksaku
  - `data/` – vstupné dáta
  - `C++/` – implementácia v jazyku C++
  - `Python/` – implementácia v jazyku Python
    - `C++/` – zdrojové súbory CUDA C++
    - `doc/` – programová dokumentácia

- `bp_text/` – text práce

- `README.md` – popis obsahu CD